Universidad Carlos III de Madrid
Escuela Politécnica Superior
Grado en ingeniería informática

Trabajo de fin de Grado

Desarrollo de una interfaz de MPI-IO para HDFS y su uso en aplicaciones Map-reduce en MPI

Autor: José Rivadeneira López-Bravo
Tutor: Félix García Carballeira

Leganés, Madrid, España
Julio 2018

Cuando un sueño aparezca, ¡Agárralo!

Larry Page

Agradecimientos

Durante esta sección me gustaría dedicar unas palabras en agradecer a todas las personas que han hecho posible este proyecto.
En primer lugar a mis padres, a mi hermana, y en general a toda mi familia, la cual me ha estado apoyando durante todos estos años, sobretodo durante la realización de este trabajo de fin de grado.

En segundo lugar me gustaría acordarme de mi tutor Félix por haberme dado la oportunidad de realizar este trabajo, el cual ha sido muy interesante y me ha permitido conocer más en profindidad tecnologías que eran poco conocidas para mí y las cuales son muy utilizadas en el mundo científico a día de hoy.

Por último me gustaría acordarme de mis amigos, aunque es cierto que durante estos últimos años apenas nos vemos ya, siempre han estado ahí para todo lo que han hecho falta.

A todos ellos muchas gracias.

Desarrollo de una interfaz MPI-IO para HDFS y su uso en aplicaciones Map-Reduce en MPI

by

José Rivadeneira López-Bravo

Abstract

Con el objetivo de resolver problemas complejos existentes en ámbitos científicos surgió la computación de altas prestaciones, la cual está formada por un sistema distribuido que se comporta de cara al usuario como un único ordenador y al que es fácil incluir potencia de cómputo. Una de las principales interfaces de HPC es la interfaz de paso de mensajes (MPI) que permite a un conjunto de máquinas sincronizar el trabajo entre ellas de una forma coordinada y distribuida.
MPI no solo permite la comunicación entre procesos, sino, que además, define una interfaz de acceso a sistemas de ficheros paralelos denominada "mpi-io".
Durante este trabajo se va a crear una nueva interfaz de mpi-io para el sistema de ficheros de HDFS. Este es un sistema de ficheros distribuido usado en el campo de BigData y desarrollado por la Apache Software foundation, el cual junto a su framework Hadoop permite el almacenaje de grandes conjuntos de datos y su posterior procesado por medio de una técnica denominada Map-Reduce. Map-Reduce, permite el procesado de datos, generando como resultado un conjunto de tuplas *(<clave>,<valor>)*. De este modo, se pueden realizar aplicaciones de Map-Reduce para determinar el número de palabras de un texto, o determinar el número de veces que se accede a una determinada web.
En este trabajo se presenta la nueva interfaz de mpi-io creada para el sistema de HDFS, y la optimización de una biblioteca Map-Reduce de MPI usando la interfaz desarrollada para Sistema de ficheros distribuido de Hadoop (HDFS) y mejorar su rendimiento añadiendo políticas de localidad.

Palabras clave: HPC BigData MPI Hadoop HDFS MapReduce

Supervisor: Félix García Carballeira
Title: Full Professor

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

Este primer capítulo presenta el proyecto realizado y cada una de las razones por las que ha sido elaborado, incluyendo las motivaciones personales (sección 1.1) , los objetivos (sección 1.2) y la estructura del documento (sección 1.3).

1.1 Motivación y desarrollo

La cantidad de datos informáticos que se generan hoy en día han ido en aumento de forma veloz durante los últimos años, llegando hasta los 30.000 gigabytes de datos por segundo que se registran actualmente. Una cifra que sigue creciendo de una forma acelerada . Todos estos datos son generados de una forma muy variada, desde aquellos que son generados por los usuarios durante su uso de internet, hasta los que son generados y tratados por las empresas. Como por ejemplo los bancos, las redes sociales o las grandes corporaciones [1].

Debido a toda esta elevada cantidad de datos, generados a diario, los medios habituales para el tratamiento y procesado de datos, como podrían ser las bases de datos relaciones y los sistemas de ficheros tradicionales, no son métodos eficientes, ya que no son capaces de tratar los datos en un tiempo admisible. Por ello surgieron una serie de técnicas denominadas Big Data.

El Big Data únicamente proporciona un conjunto de técnicas distribuidas, las cuales permiten tratar todo el conjunto de datos generados a día de hoy en un tiempo admisible. Una de las empresas pioneras en este concepto de fue Google [2], que diseñó su propio sistema de ficheros distribuido con el nombre de Google File System (GFS) [3] junto con su framework Map-Reduce [4].

Tras la iniciativa de Google y al ver los buenos resultados obtenidos, la comunidad de software libre, se lanzó a crear su propio sistema de ficheros distribuido, así como su propio framework de Map-Reduce. De este modo la Apache Software Foundation desarrolló junto a Yahoo su propio framework denominado Hadoop, así como su sistema de ficheros distribuido al cual llamaron “HDFS"[5].

Además de las técnicas anteriores de Big Data, existe otro conjunto de técnicas denominado computación de altas prestaciones (HPC). Esta técnica es usada principalmente por científicos para realizar cálculos matemáticos que deben ser realizados de una manera paralela con el objetivo de obtener los resultados lo más rápido posible. La principal ventaja que ofrece la HPC es permitir añadir nuevos ordenadores o estaciones de trabajo a un cálculo que se quiera realizar, de manera que cada uno de estos ordenadores colaboren en el resultado final. Para ello, cada uno de ellos suma sus recursos (procesador, memoria) consiguiendo reducir los tiempos de cómputo necesarios.

Dentro de la técnica destaca la Interfaz de paso de mensajes (MPI), la cual es una librería usada en sistemas distribuidos y con varios procesadores, la cual permite lanzar un programa a varios ordenadores simultáneamente y que se coordinen entre ellos, de manera que cada una de las máquinas aporte una pequeña parte del resultado final de la aplicación. Además, MPI no solo define operaciones de coordinación entre los procesos, sino que además, tiene definida una interfaz de entrada y salida que facilita el trabajo con varios sistemas de ficheros diferentes. Esta interfaz se denomina MPI-IO y está contenida dentro de MPI [6].

Dentro de la interfaz, se define el formato estándar de las funciones de la biblioteca de paso de mensajes, también define el estándar para comunicaciones punto a punto, comunicaciones colectivas... etc [6]. Como consecuencia, no solo hay una implementación de MPI si no que hay varias implementaciones, como podría ser la implementación de Openmpi o bien la de MPICH.

En este trabajo se va a presentar una forma de unir ambas tecnologías por medio de una interfaz del sistema de ficheros. Para ello, en primer lugar, se va a dar soporte a MPI para que pueda trabajar sobre el sistema de ficheros de HDFS. En segundo lugar, se va a proceder al uso de técnicas de Big Data, en aplicaciones MPI, usando esta nueva interfaz creada.

1.2 Objetivos

En este proyecto se presentan dos objetivos principales; el primero de ellos consiste en la **elaboración de una interfaz de MPI sobre el sistema de ficheros HDFS** con el objetivo de poder

usar aplicaciones de HPC con un sistema de ficheros distribuido, como es HDFS. El segundo objetivo principal es la **ejecución de aplicaciones de HPC sobre HDFS y analizar su rendimiento, en concreto, en la ejecución de aplicaciones de** Map-Reduce en MPI.
Además de este objetivo principal, se persiguen una serie de objetivos secundarios, los cuales son detallados en la siguiente lista:

- Ofrecer una comparativa entre el uso de la interfaz de MPI, para lectura y escritura; y el uso de la interfaz nativa.
- Mejorar el rendimiento de operaciones de Map-Reduce utilizando una librería de MPI.
- Modificar una librería de Map-Reduce para adaptar su funcionamiento a MPI con HDFS.
- Opimizar una librería e Map-Reduce basada en MPI (denominada Mimir) para proporcionar localidad en el acceso a los datos.
- Comparar los tiempos obtenidos de Map-Reduce por ambos métodos (MPI y Hadoop).

1.3 Estructura del documento

El presente documento se encuentra estructurado en los siguientes capítulos:

- Capítulo 1 *Introducción*: en este capítulo se realiza una breve introducción a el trabajo realizado y una pequeña referencia que sigue el documento.
- Capítulo 2 *Estado del arte*: en este capítulo se realiza una descripción del estado en el que se encuentran las tecnologías disponibles y las cuales han sido usadas en la elaboración de este Trabajo de fin de grado (TFG).
- Capítulo 3 *Análisis*:en este capítulo se presenta una descripción del proyecto, se explica la solución elegida y se define el conjunto de requisitos. Además, se explica el marco regulador de este proyecto.
- Capítulo 4 *Diseño*: en este capítulo se detalla el diseño del sistema y de todos y cada uno de sus componentes.
- Capítulo 5 *Implementación y desarrollo*: en este capítulo se detalla como ha sido implementada la interfaz y las modificaciones realizadas en la librería.
- Capítulo 6 *Verificación validación y evaluación*: en este capítulo se detalla la verificación y validación del proyecto. También se incluye una evaluación de rendimiento usando la interfaz desarrollada.
- Capítulo 7 *Planificación y costes*: en este capítulo se detalla la planificación del proyecto además, se realiza un presupuesto del mismo. Por último, se describe el entorno socioeconómico.

- Capítulo 8 *Conclusiones y trabajo futuro*: en este capítulo se detallan las conclusiones obtenidas del proyecto y se definen las vías de investigación futuras.

Capítulo 2

Estado del arte

En este capítulo se detallará en qué punto se encuentra la tecnología utilizada para el desarrollo de este proyecto. Como se ha indicado en la introducción, en este proyecto se mezclan dos tecnologías básicas; por un lado tenemos el Big Data (sección 2.1) y por el otro tenemos la HPC (sección 2.2), en el último punto de este capítulo se detallan un conjunto de librerías que permiten ejecutar técnicas de Big Data en aplicaciones de HPC (sección 2.3).
Este proyecto se engloba en el uso de sistemas distribuidos, por este motivo, en primer lugar, se definirá el concepto de sistema distribuido. Esta definición varía según el libro que se consulte, aunque todos tienen rasgos similares, de tal modo que, podemos concluir que un sistema distribuido se puede definir como un conjunto de máquinas conectadas a una red, las cuales se comunican y sincronizan entre ellas por medio del paso de mensajes [7]; o bien, como una colección de ordenadores independientes, los cuales el usuario percibe como un único sistema [8].
Todos los sistemas distribuidos presentan una serie de características comunes:

1. Homogéneo: las diferencias entre cada uno de los ordenadores se encuentran escondidas al usuario.

2. Escalable: debe ser fácilmente expansible.

3. Tolerante a los fallos: el fallo en uno de los ordenadores no debe afectar al resto.

4. Se debe comportar como un único sistema.

La figura 2-1 muestra el estado actual de las dos tecnologías que se presentarán posteriormente. La parte izquierda de la imagen se corresponde con los componentes de Big Data, mientras que la parte derecha se corresponde con cada uno de los componentes que conforman el HPC.

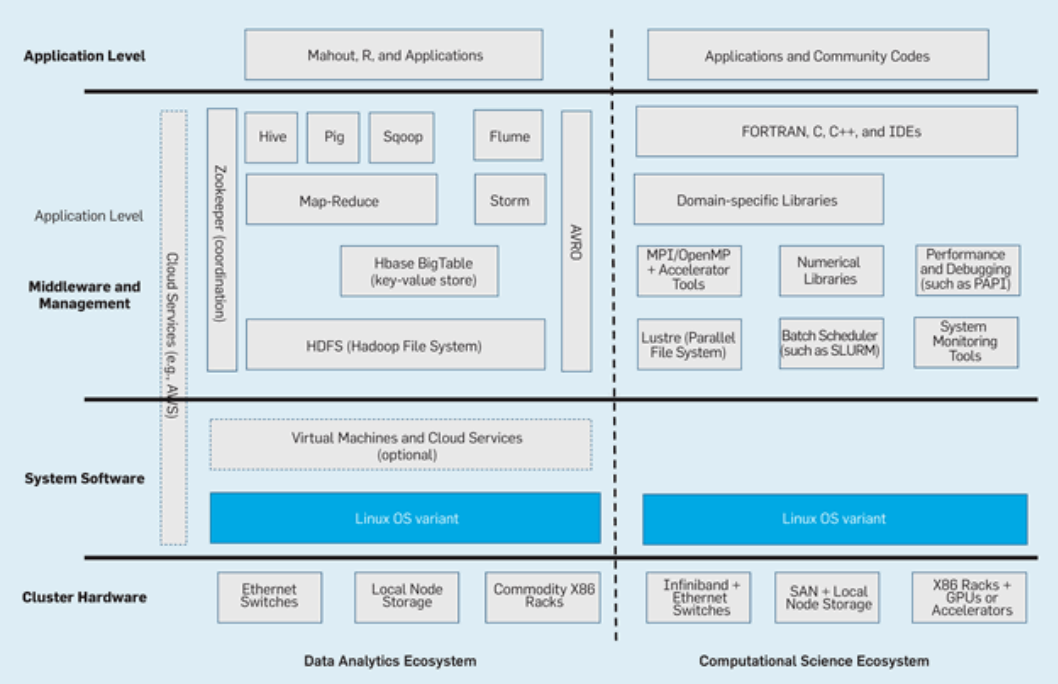

Figura 2-1: *Estructura Big Data y HPC*

2.1 Big Data

La definición de Big Data corresponde a un conjunto de datos de grandes dimensiones (del orden de petabytes) que debido a su diversidad, a su distribución y al tiempo que requiere su procesado no se pueden procesar por medio de las técnicas habituales, es decir, no se pueden procesar con las bases de datos relacionales. Estos datos tampoco se pueden almacenar en los sistemas de ficheros disponibles actualmente.
Consecuencia de estas características se tuvieron que diseñar nuevas arquitecturas, algoritmos y herramientas para su procesado, de tal modo que se pudiera hacer en un tiempo admisible [9] [10].

La cantidad de datos generados hoy en día, tanto por las empresas como por los usuarios, presenta una serie de problemas que deben ser tenidos en cuenta al tratarse. Estas dificultades son principalmente 4:

- Heterogeneidad: los datos generados son muy variados y pueden estar incompletos.
- Escalado: debido a la gran cantidad de datos que se deben almacenar, los sistemas de ficheros deben ser escalables, pudiendo aumentar su tamaño sin que provoque una pérdida en el rendimiento del sistema.
- Rendimiento: el sistema debe proporcionar una gran velocidad de escritura y de lectura

de los datos almacenados.

- Privacidad: debido a las regulaciones de privacidad, los datos deben estar lo suficientemente seguros y no comprometer la seguridad de las personas.

Por todos estos problemas descritos anteriormente, Google se vió en la necesidad de crear un nuevo sistema de ficheros denominado GFS, el cual tenía las siguientes características principales: [3].

- Es un sistema de ficheros distribuido.
- Se encuentra dotado de tolerancia a fallos.
- El tamaño de bloque es de grandes dimensiones del orden de 128 MB, con el fin de aumentar el rendimiento del sistema.
- Tiene un gran rendimiento devolviendo las peticiones realizadas simultáneamente por gran cantidad de usuarios.

En la figura 2-2 se muestra la arquitectura del GFS. Como se observa hay un nodo máster en el cual se almacena toda la información del sistema de ficheros, nombre de los ficheros, árbol de directorios y en que nodo se almacena cada uno de los ficheros. Por otro lado, se encuentran unos nodos denominados *chunkserver*, en los cuales se almacenan los bloques de los ficheros.

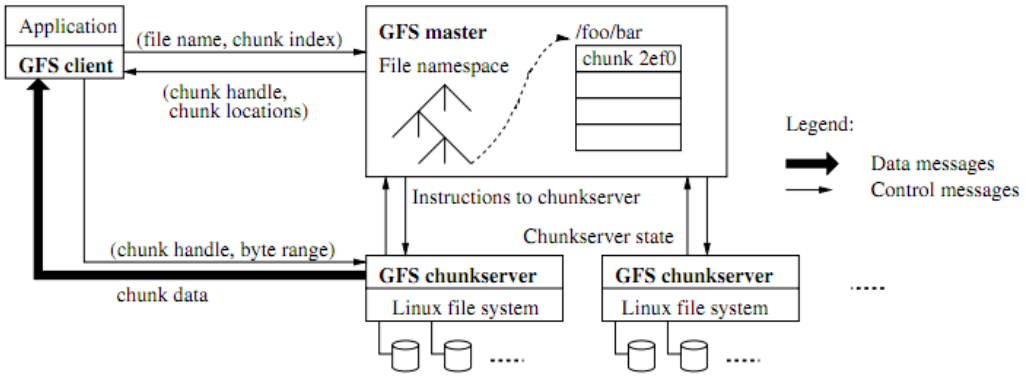

Figura 2-2: *Arquitectura del sistema de ficheros GFS*

Una vez que Google consiguió tener su sistema de ficheros, necesitaba un algoritmo con el que conseguir procesar todos los datos almacenados de una manera eficiente. Para ello diseño su algoritmo de Map-Reduce.

2.1.1 MapReduce

Es un técnica de procesado de datos que trata grandes conjuntos de datos en un tiempo admisible, generando un nuevo conjunto de datos como salida. El funcionamiento de una operación de Map-Reduce es totalmente paralelo y por ello muy eficiente, debido a que ejecuta el mismo programa en cientos de máquinas de forma simultánea.
Esta técnica se basa en dos funciones principalmente, denominadas Map y Reduce. Estas dos funciones son programadas por el usuario de tal modo que se pueden realizar infinidad de programas, como por ejemplo el contar el número de veces que aparece una palabra en un documento, o bien conocer el número de accesos que tiene una determinada URL.
La mejor manera de explicar la función de Map-Reduce es por medio del ejemplo de contar palabras[4].

Map

La función de Map recibe como entrada un gran conjunto de datos, por cada palabra encontrada, genera una nueva tupla de salida con la siguiente forma:

<Clave,Valor>

Donde la *clave* es cada una de las ocurrencias que haya encontrado en el conjunto de datos de entrada. En el ejemplo de contar palabras, el valor se correspondería con "1".
Un pseudocódigo de la función de Map sería:

```
0  map(String key, String value) {
1      //Key:Nombre del documento
2      //value: Contenido del documento
3      for (cada palabra p en value){
4          EmitirResultadoIntermedio(p,"1");
5      }
6  }
```

Reduce

La función de Reduce recibe el conjunto de datos generado por la función de Map, es decir, recibe tuplas del tipo *<clave,1>*. A partir de dichas entradas, las agrupa por clave y genera la salida, la cual son tantas tuplas como claves diferentes hubiera en el texto de entrada de la función Map que se detalló en la sección 2.1.1.

<Clave,Suma de todos los valores asociados a esa clave>

Un pseudocódigo de la función de Reduce sería:

```
0       reduce(String key, Iterator value) {
1               //Key: una palabra
2               //values: Ocurrencias de la clave
3               for (cada ocurrencia v en value){
4                   resultado = resultado + value;
5               }
6               EmitirResultadoFinal(String(value));
7       }
```

2.1.2 Hadoop

Basándose en la idea original de Google la Apache Software Foundation [11], creó un proyecto de software libre denominado Hadoop. Este es un framework escrito en java que permite el procesamiento distribuido de grandes conjuntos de datos a través de un Cluster de cientos de ordenadores usando modelos simples de programación. Está diseñado para funcionar en cientos de máquinas e incluye tolerancia a fallos [12].
Cabe destacar que, uno de los grandes contribuidores al desarrollo de este framework fue la empresa de Yahoo [13], la cual era y sigue siendo una de las principales empresas que usan este framework para el procesado de los datos.
Hadoop tiene dos características principales; su propio sistema de ficheros denominado HDFS, definido en la sección 2.1.3 de este capítulo; y su propio Hadoop MapReduce, definido en la sección 2.1.1. [5] [10]

2.1.3 HDFS

HDFS es el sistema de ficheros usado por Hadoop, este sistema de ficheros presenta unas características muy similares al GFS. Además, la semántica de cada uno de los ficheros (permisos y almacenamiento) es similar a la usada por el sistema de ficheros de Unix [14].

Arquitectura

La arquitectura del sistema de ficheros HDFS consta de varios componentes, descritos en el artículo [5] . En el presente documento ofreceremos una descripción de los más importantes.

- Datanode: en los *Datanode* es donde se almacenan los bloques de datos de cada uno de los ficheros, cuyo objetivo es tener una alta tolerancia a los fallos y además de conseguir un mayor rendimiento en operaciones de lectura en los ficheros. Cada uno de los bloques de un fichero se encuentra almacenado simultáneamente en varios *Datanode*.
- Namenode: almacena toda la información correspondiente con el sistema de ficheros. Esto son los meta-datos de cada uno de los ficheros en los cuales se incluye: su nombre,

su tamaño, los permisos de lectura y escritura que dispone. Además, incluye una lista de en que *Datanode* se encuentra almacenado cada uno de los bloques que forman parte de ese fichero. Cabe destacar que en cada distribución de Hadoop solo existe un *namenode*.

- Secondary NameNode: en este componente se almacena una copia del *NameNode*, además de un log de todos los cambios que se han hecho en el sistema de ficheros. De este modo, en el caso de que el sistema de ficheros o el *Namenode* se corrompa, se puede recuperar la información de todo el sistema de ficheros.

En la figura 2-3, se detalla la arquitectura de forma gráfica [15].

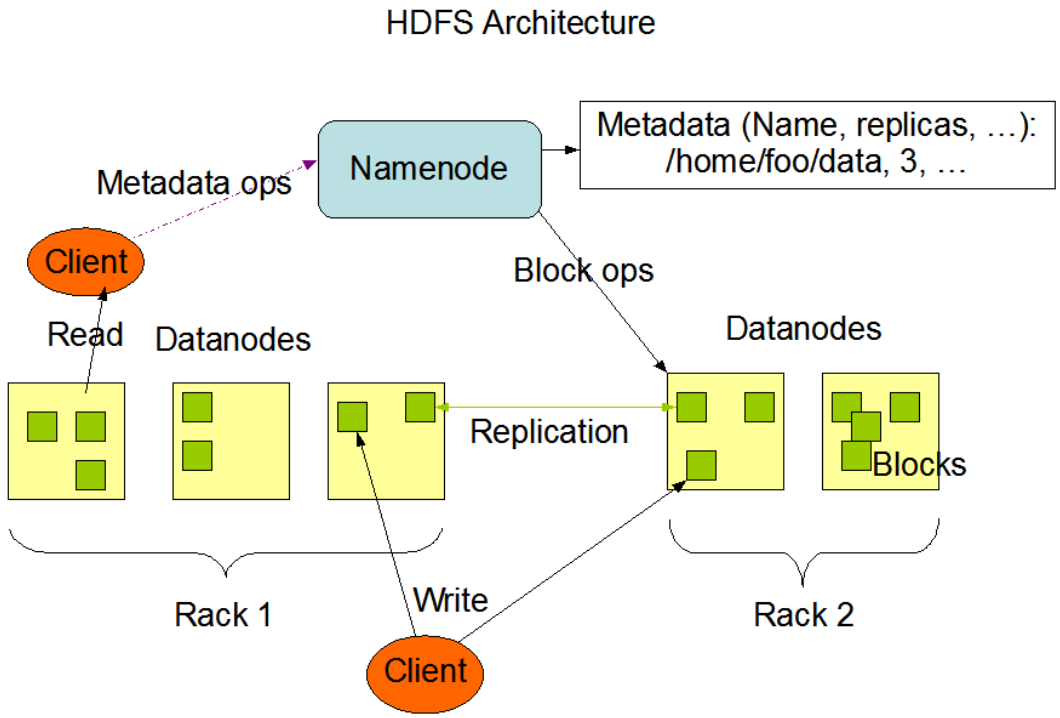

Figura 2-3: *Arquitectura del sistema de ficheros HDFS*

Limitaciones del sistema de ficheros

Este sistema de ficheros presenta una serie de limitaciones debido a su implementación. Hadoop sigue el modelo de un solo escritor y varios lectores, esto quiere decir que solo se permite un proceso escribiendo en un fichero simultáneamente, debido a que cuando un proceso abre un fichero en modo escritura, el *Namenode* bloquea el fichero, evitando que otros procesos puedan abrirle en ese mismo modo. Además cuando un bloque ha sido escrito en el sistema de ficheros de HDFS, no se permite su modificación. La limitación anterior no sucede con las lecturas en las cuales puede haber más de un proceso de lectura a la vez. La figura 2-4 muestra cómo es el proceso de escritura de un fichero en HDFS, en primer lugar el cliente crea el path

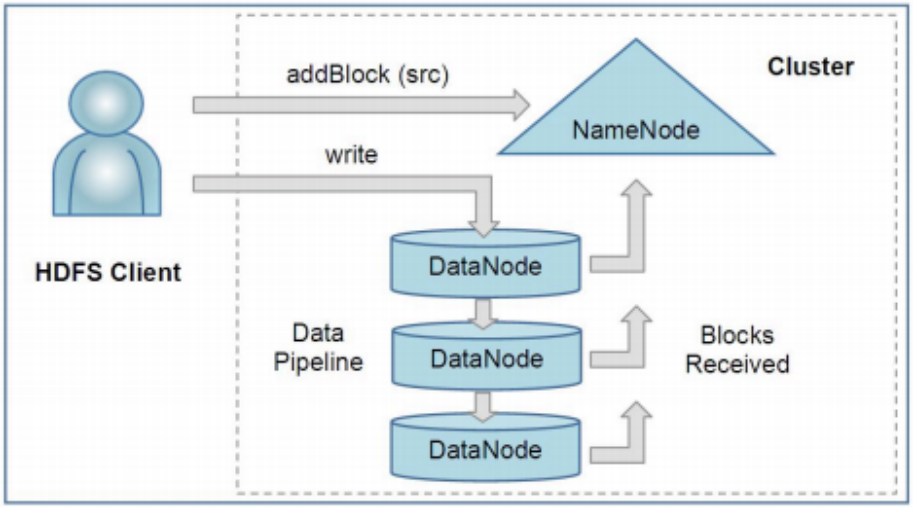

Figura 2-4: *Proceso de escritura de un fichero en HDFS*

en el *Namenode*, entonces este devuelve una lista de *Datanodes* en los que almacenar los bloques, tras ello se escribe en un datanode y este va propagando las escrituras al resto en forma de pipeline. [5]

2.1.4 Hadoop MapReduce

Uno de los principales usos de Hadoop es su uso en aplicaciones de Map-Reduce. Estas son similares a las que diseñó Google que fueron detalladas en la sección 2.1.1. Como Hadoop se encuentra escrito en el lenguaje de programación de java, para programar cualquier programa de Map-Reduce dentro de este entorno, se debe utilizar ese lenguaje de programación. Este framework no dispone de interfaces para otros lenguajes de programación, por esto se encuentra limitado al lenguaje de programación indicado anteriormente.

En la práctica, el uso de funciones de Map-Reduce, usando este framework, consigue un alto rendimiento procesando datos. Así,Yahoo consiguió procesar 1000 Terabytes de datos a través de 3500 nodos en unas 16 horas aproximadamente. [5].

2.2 Computación de altas prestaciones

La computación de altas prestaciones consiste en la práctica de agregar poder de cómputo con el objetivo de obtener mayor rendimiento. Para ello, se unen ordenadores en red, de modo que cada uno de ellos sume su potencia de cómputo, y sus recursos (Ram, CPU) al resto. [16]

El principal uso de la HPC se da en ámbitos científicos, debido a que, suele ser necesario un gran poder de cálculo con el objetivo de resolver ecuaciones complejas en un tiempo finito.

2.2.1 MPI

MPI es una biblioteca de paso de mensajes utilizadas para construir aplicaciones en HPC, esta se utiliza principalmente en el campo de la ingeniería y en la resolución de ecuaciones científicas. [6] [17]
Las aplicaciones escritas en MPI siguen el modelo de programación paralela, basada en el paso de mensajes. En este modelo los datos son enviados del espacio de memoria de un proceso a otro a través de operaciones colectivas. Estas operaciones colectivas pueden ser el envío de mensajes de un proceso a cualquier otro, el envío de un mensaje de un proceso a cada uno de los procesos restantes (Broadcast), o el envío de los datos de cada uno de los procesos a un proceso (Gather). Además, dentro de la especificación de MPI, se define una interfaz para acceder a los sistemas de ficheros de una manera compartida, es decir, hace referencia a la interfaz de MPI-IO, detallada en la sección 2.2.2. MPI presenta dos características básicas para su funcionamiento, la primera de ellas consiste en la asignación de un identificador básico a cada uno de los procesos que está ejecutando la aplicación, que va desde el valor 0 hasta el proceso n-1. Además, MPI, para la comunicación de los procesos, usa *comunicadores* que sirven para indicar qué procesos tienen que formar parte de una operación colectiva. Cuando unos procesos llaman a una operación colectiva, el resto de procesos esperan a que todos los procesos que están dentro del comunicador indicado en la llamada, ejecuten la función.
Como se ha indicado anteriormente MPI solo define la interfaz, por ello, hay diferentes implementaciones de MPI en el mercado, las más conocidas son open-mpi [18] y MPICH [19]. Esta última, además, es utilizada en 9 de los 10 supercomputadores más potentes del mundo [20].

MPICH

Esta implementación de MPI va a ser utilizada durante este TFG. Fue desarrolla por el laboratorio Argonne en los Estados Unidos de América, debido a la necesidad de hacer una implementación portable y de alto rendimiento de MPI. Además, fue desarrollada como un proyecto de software libre.

2.2.2 MPI-IO

Además de definir una interfaz de paso de mensajes MPI, también incluye una interfaz para el acceso a ficheros dentro de la propia MPI denominada MPI-IO. Las funciones que define esta interfaz se pueden agrupar en tres grupos, como se detalla en el capítulo 13 de el manual de usuario [6].

Funciones de manipulación de los ficheros

Estas funciones permiten la interacción con el sistema de ficheros, ya sea abriendo un fichero, cerrándolo o eliminándolo; además del acceso a los metadatos de los ficheros. Las principales funciones que MPI-IO, define para estas funcionalidades, son las siguientes:

- MPI_File_open: esta función es la encargada de abrir un fichero. Todas las llamadas a están función son colectivas, por lo tanto, todos los procesos que se encuentran en el comunicador tienen que realizar la llamada.
- MPI_File_close: esta función cierra un fichero, al igual que la función de apertura, también es colectiva; por ello, los procesos se bloquean esperando a que todos los que abrieron el fichero lo cierren.
- MPI_File_get_size: esta función permite obtener el tamaño de un fichero.
- MPI_File_set_size: esta función permite redimensionar un fichero.
- MPI_File_delete: permite eliminar un fichero dentro del sistema de ficheros.
- MPI_File_size: permite el desplazamiento del puntero interno de un fichero.

Funciones para definir vistas en los ficheros

Definir una vista en un fichero dentro de MPI-IO consiste en indicar los bytes del fichero que son accesibles para ese proceso. MPI define una serie de funciones con el objetivo de poder definir vistas a los ficheros. La principal función de definir vistas a los ficheros es que se permite de este modo un acceso no secuencial al mismo. Las principales funciones para el uso de las vistas en MPI son:

- MPI_File_set_view : la cual permite definir una vista en un fichero.
- MPI_File_set_view : la cual permite obtener una vista de un fichero.

Funciones de acceso a los ficheros

Las funciones que se indican en este apartado, son aquellas que permiten el acceso a los datos. Por lo tanto, son las realmente importantes, ya que definen los prototipos de las funciones tanto de lectura como de escritura en los ficheros. La tabla 2.1 muestra todas las funciones que MPI-IO tiene definidas:

Posicionado	Sincronización	Coordinación	
		No colectiva	Colectiva
Offset explícito	Bloqueante	MPI_File_read_at MPI_File_write_at	MPI_File_read_at_all MPI_File_write_at_all
	No bloqueante	MPI_File_iread_at MPI_File_iwrite_at	MPI_File_iread_at_all MPI_File_iwrite_at_all
	División colectiva	N/A	MPI_File_read_at_all_begin MPI_File_read_at_all_end MPI_File_write_at_all_begin MPI_File_write_at_all_end
Puntero individual	Bloqueante	MPI_File_read MPI_File_write	MPI_File_read_all MPI_File_write_all
	No bloqueante	MPI_File_iread MPI_File_iwrite	MPI_File_iread_all MPI_File_iwrite_all
	División colectiva	N/A	MPI_File_read_all_begin MPI_File_read_all_end MPI_File_write_all_begin MPI_File_write_all_end
Puntero compartido	Bloqueante	MPI_File_read_shared MPI_File_write_shared	MPI_File_read_ordered MPI_File_write_ordered
	No bloqueante	MPI_File_iread_shared MPI_File_iwrite_shared	N/A
	División colectiva	N/A	MPI_File_read_ordered_begin MPI_File_read_ordered_end MPI_File_write_ordered_begin MPI_File_write_ordered_end

Cuadro 2.1: *Tabla de funciones de acceso a los datos.*

- Posicionamiento: el posicionamiento hace referencia al tipo de puntero que usará la función. De este modo, en MPI-IO, se puede elegir entre tres tipos de punteros diferentes:

  1. Offset explicito: en este caso cada fichero leerá únicamente la porción del fichero que se le indique como parámetro. En estas funciones, el puntero interno que lleva cada uno de los descriptores de fichero, no es modificado.
  2. Puntero individual: dentro de cada uno de los identificadores de un fichero se encuentra un puntero, este indica en que posición del fichero se encuentra el proceso, así cada una de las lecturas actualiza la posición de este.
  3. Puntero compartido: al igual que en el caso anterior, dentro del descriptor de cada uno de los ficheros se encuentra un puntero. La diferencia es que en este caso, el puntero es compartido por cada uno de los procesos. De este modo la lectura de uno de ellos lo actualiza en todos.

- Sincronización:

1. Bloqueante: una función se dice que es bloqueante, cuando el proceso se bloquea hasta que la llamada a la función se ha completado.
2. No bloqueante: una función se dice que no es bloqueante cuando el proceso no se bloquea y se sigue ejecutando al realizar la llamada. en este caso MPI-IO, tiene un mecanismo para indicar cuando una llamada a una función no bloqueante ha finalizado.
3. Particionado colectivo: estas funciones son igual que las funciones no bloqueantes pero, con la diferencia de que se encuentran divididas en dos: una función de begin, que inicia la operación y; otra función de end, que es la encargada de finalizar.

- Coordinación:

1. Colectiva: estas funciones indican que todos los procesos que han abierto el fichero, deben llamar a la función de escritura o de lectura. Siempre que se pueda; se deben utilizar estas funciones, porque internamente están más optimizadas que aquellas que no son colectivas.
2. No colectiva: las funciones no colectivas no tienen por qué ejecutarlas cada uno de los procesos, sino que la llamada que haga un proceso es independiente de las del resto.

Una vez que se ha detallado cómo es la definición de la interfaz, se va a hablar de la implementación de MPI-IO, más en concreto de la implementación usada por MPICH y de la que se habló en la sección 2.2.1. Esta tiene dos componentes principales denominados Romio y ADIO.

Romio

Romio es una implementación portable y de alto rendimiento de MPI-IO. La última versión que se encuentra disponible implementa todas las funciones definidas en el estándar MPI, a excepción del soporte para la interoperabilidad de los ficheros, así como la definición por parte de los usuarios de manejadores de error para los ficheros. [21]
Romio implementa dos importantes técnicas para la optimización de la entrada y salida, el uso de estas técnicas consigue un mayor rendimiento en las aplicaciones que ejecutan Romio. La primera de ellas se denomina *data sieving* [22], técnica que permite el acceso a partes no contiguas de un fichero de una forma eficiente. Para ello agrupa, muchas operaciones de entrada y salida pequeñas, en una sola operación grande de entrada y salida. La segunda optimización implementada por Romio se denomina *two-phase I/O* [23], esta optimización solo se aplica a llamadas colectiva, y consiste en definir un conjunto de agregadores que serán los que accedan

al fichero y luego estos dividen los datos entre el resto de procesos que forman parte de la operación colectiva.

ADIO

ADIO (Abstract Device Input Output) es una estrategia portable y eficiente que permite implementar interfaces para sistemas de ficheros paralelos de una manera sencilla. ADIO consiste en un pequeño conjunto de funciones de entrada y salida paralela, estas funciones comprenden desde la apertura de un fichero hasta las operaciones de lectura y escritura de los mismos. Cada una de estas funciones que define ADIO, deben ser implementadas de una manera diferente para cada sistema de ficheros.
ADIO está implementado dentro de MPICH, de modo que, la inclusión de soporte para el acceso a un nuevo sistema de ficheros dentro de MPICH se puede hacer incluyendo este nuevo sistema de ficheros dentro de ADIO. [24]

2.3 Map Reduce con MPI

En la actualidad, se están desarrollando bibliotecas que permitan aplicar operaciones de Big Data en entornos de HPC. Algunas de estas librerías permiten la realización de aplicaciones de Map-Reduce utilizando funciones de MPI y de su subinterfaz Interfaz de paso de mensajes, aplicada al acceso a los sistemas de ficheros (MPI-IO). Durante esta sección, se presenta la librería Mimir [25], librería permite a los usuarios la programación de sus propias funciones de Map y Reduce, y que se podrán ejecutar en entornos de MPI. Su implementación es una mejora de la una librería similar denominada: mr-mpi [26].
Las dos librerías anteriores se basan en el concepto de *páginas*, estas páginas son un espacio de memoria donde se almacenan los conjuntos de clave valor. La principal diferencia de estas dos bibliotecas radica en cómo gestionan cuando una página de memoria se llena. En mr-mpi cuando una página se llena el proceso la escribe en disco, mientras que en Mimir, cuando una página se llena se le envía a otros procesos y en cada uno de ellos se realizan operaciones de Reduce. De este modo la librería de Mimir obtiene un mejor rendimiento.
las funciones de Map-Reduce programadas en la librería de Mimir son muy similares a las programadas usando Hadoop. En primer lugar se mostrarán las funciones de Map-Reduce usando el framework de Hadoop.

```
map(LongWritable key, Text value, Context context) {
        String line = value.toString() // Siguiente linea del fichero
        StringTokenizer tokenizer = new StringTokenizer(line);
        while(tokenizer.hasMoreToken()){
            word.set(tokenizer.nextToken()); //Siguiente palabra
            context.write(word,1);//Almacena la palabra (clave) y un uno (valor)
        }
}
```

```
reduce(Text key, Iterable<IntWritable> values,Context context) {
        //Key la clave
        //values ocurrencias de las clave
        int sum = 0;
        for (IntWirtable val:values){ //Suma todas las veces que aparece la clave
            sum = val.get();
        }
        Context.write(key,newIntWirtable);
}
```

A continuación, se detalla el código de las funciones de Map y Reduce usando la librería de Mimir.

El siguiente fragmento muestra la función de Map usando la librería de Mimir.

```
map(Readable<char*,void> * input, Writable<char*,uint64_t>*output, void * ptr) {
        //input->texto de entrada
        //output->Salida de clave valor
       char * line = NULL;
       while(input->read(line,NULL)){
            //Por cada una de las lineas del fichero
            char * saveptr = NULL;
            char * word = strtok_r(line," ",&saveptr);//Cogemos la siguiente palabra.
            while(word != NULL){
                output->write(&word,1); //Generamos conjunto clave valor
                word=strtok_r(NULL," ",&saveptr);//Leemos la siguiente palabra
            }
        }
}
```

El siguiente fragmento muestra la función de Reduce usando la librería de Mimir.

```
0   reduce(Readable<char*,void> * input, Writable<char*,uint64_t>*output, void * ptr) {
1           char * key = NULL;
2           unit64_t val = 0;
3           uint64_t count = 0;
4           while(input -> read(&key,&val) == true){
5               //Sumamos todos los valores asociados a una clave
6               count += val;
7           }
8           output->write(&key,&count);//Generamos la salida de la funcion reduce
9   }
```

Como se aprecia en el código, las funciones de Mimir (Map y Reduce) se programan de una manera similar a las realizadas en el entorno de Hadoop.

Capítulo 3

Analisis

Durante este capítulo se analiza el proyecto que se ha llevado a cabo. En la primera sección, se realiza una descripción del proyecto realizado (sección 3.1). En la segunda sección, se analizan las diferentes opciones disponibles para la realización de dicho trabajo y se detalla por que se ha escogido cada una de ellas (sección 3.2). Para finalizar, se analizan los requisitos de usuario que se han tenido en cuenta durante el desarrollo de este TFG (sección 3.3).

3.1 Descripción del proyecto

Como se ha comentado en el capítulo 1 del presente TFG, este consta de dos grandes objetivos. El primero de ellos consiste en la elaboración de una interfaz MPI-IO para el sistema de ficheros HDFS, consiguiendo la utilización de aplicaciones de MPI que permitan la interacción con el sistema de ficheros de HDFS y en segundo lugar, se va a proceder al uso de una librería de Map-Reduce en MPI para analizar ficheros almacenados dentro del sistema de ficheros de HDFS, para ello se va a modificar esta librería para permitir el su uso con MPI-IO (la versión inicial solo usa Portable Operating System Interface (Posix)). A continuación se va a optimzar esta librería para incorporar localidad en el acceso a los datos. En la figura 3-1 se puede observar el estado inicial del proyecto, indicado anteriormente en el capítulo 2. Como se observa en la figura, las aplicaciones de MPI no pueden acceder al sistema de ficheros de HDFS debido a que la interfaz no admite operaciones sobre este sistema de ficheros.

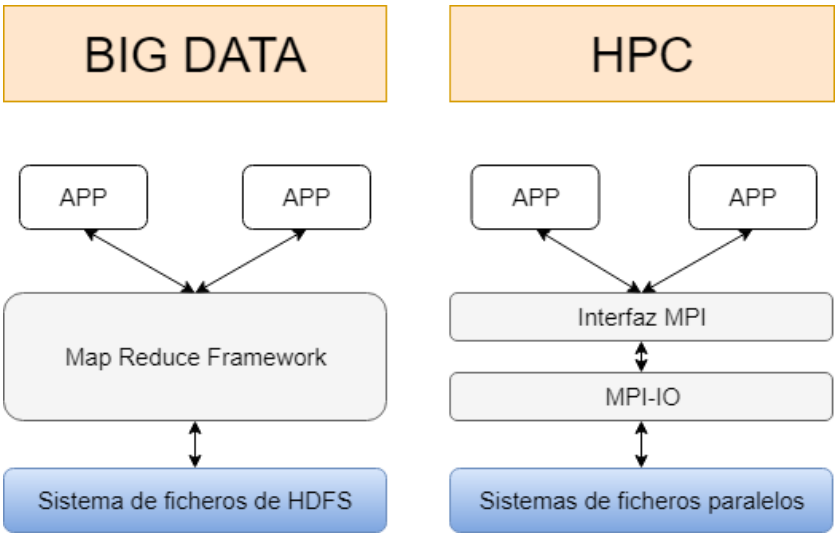

Figura 3-1: *Estado inicial del proyecto*

Con el objetivo de que las aplicaciones escritas en MPI puedan operar con HDFS, se extiende la interfaz de MPI-IO. Para crear dicha extensión se tiene que crear un conector de HDFS dentro de la interfaz ADIO de MPI-IO. Una vez implementada la interfaz, las aplicaciones de MPI ya pueden acceder al nuevo sistema de ficheros. La figura: 3-2, muestra el modo en el que las aplicaciones de MPI pueden accceder al sistema de ficheros de HDFS.

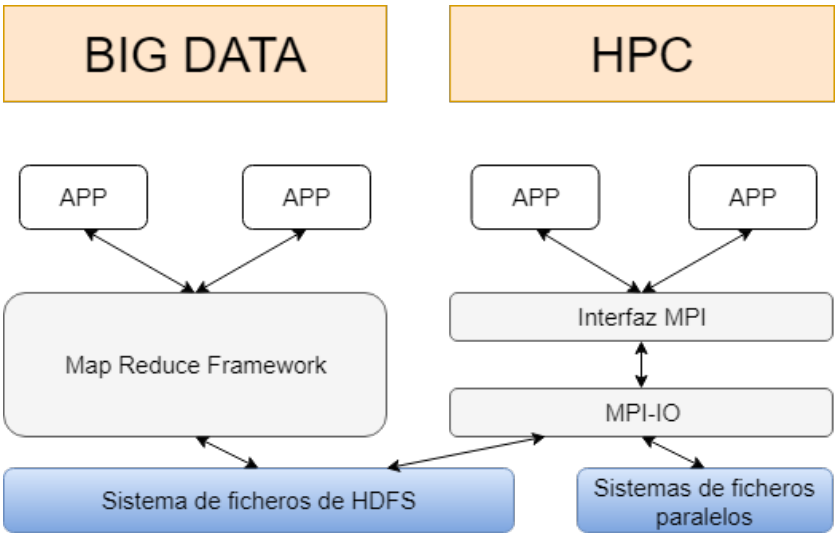

Figura 3-2: *Extensión de la interfaz para HDFS*

Tras la ampliación de la interfaz de MPI-IO, se introduce un nuevo componente, una librería de Map-Reduce. Este nuevo elemento debe permitir la apertura de ficheros de HDFS a través de la interfaz de MPI. El objetivo que se persigue introduciendo este componente consiste en la realización de pruebas de Map-Reduce utilizando la interfaz de MPI y, además, usando el sis-

tema de ficheros de HDFS. Debido a que HDFS es un sistema de ficheros distribuido, no todas las máquinas tienen almacenados todos los bloques de datos, motivo por el cual se introduce el concepto de localidad. Esto indica que cada bloque de datos será procesado por algún proceso que se encuentre ejecutando en la máquina en la que se encuentra almacenado dicha porción del fichero.

En la figura 3-3 se observa el estado final del proyecto y cómo se unirán cada uno de los diferentes componentes.

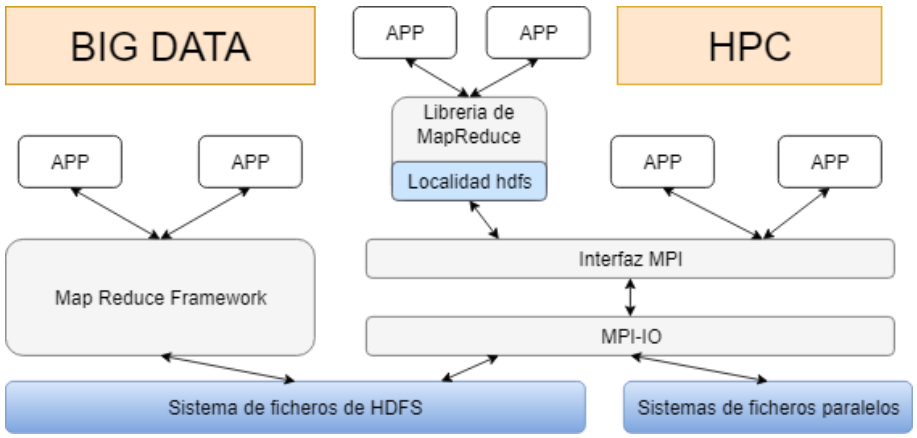

Figura 3-3: *Estructura Big data y HPC*

3.2 Elección de la solución

Durante esta sección se analizarán, en detalle, cada una de las diferentes tecnologías que fueron consideradas a la hora de realizar el presente trabajo y, el porqué de su elección. En un primer lugar, se ofrece una comparativa entre las diferentes implementaciones que se han barajado de MPI y a la cual se le incluirá el acceso al sistema de ficheros de HDFS. Estas son: MPICH [19][27], open-mpi [18] y IBM Spectrum [28]. El siguiente cuadro 3.1 muestra una comparativa entre las diferencias más importantes que presentan cada una de las implementaciones.

Implementación	MPICH	OpenMPI	IBM Spectum
Lenguaje interfaz C/C++	Si	Si	Si
Software libre	Si	Si	No
Soporta último estándar	Si	No	Si
Soporte ABI	Si	No	No
Soporta threads	Si	No	Si

Cuadro 3.1: *Comparativa de las implementaciones de MPI*

- Lenguaje interfaz: indica los lenguajes que soporta la interfaz.
- Software libre: indica si es un proyecto de software libre , por el contrario es un software propietario.
- Soporte último estándar: tiene soporte al último estándar de MPI.
- Soporte ABI: las aplicaciones compiladas con versiones superiores, se podrán ejecutar con versiones inferiores [29].
- Soporta threads: la implementación soporta la creación de threads internos.

Según el cuadro anterior 3.1 se ha decidido usar la implementación de MPICH debido, principalmente a que se trata de una implementación de software libre, a que soporta el último estándar y además, que permite la implementación de threads. Por otro lado se analiza que librería de Map-Reduce se utilizará sobre MPI. En este caso, se disponía de dos opciones muy similares, las librerías de Mimir y mr-mpi.

Implementación	Mimir	Mrmpi
Lenguaje	C/C++	C/C++
Optimización IO	Si	No
Sistemas de ficheros no Posix	No	No
Elección de las funciones de map y de reduce	Si	Si
Software libre	Si	Si

Cuadro 3.2: *Comparativa de librerias MAP-REDUCE sobre MPI*

- Lenguaje: indica los lenguajes que soporta la interfaz.
- Optimización IO: hace referencia a si se encuentran optimizadas las operaciones de entrada y salida o, si por el contrario hace mucho uso de acceso a disco.
- Sistemas de ficheros no Posix: soporta la ejecución de Map-Reduce sobre ficheros que no siguen el estándar Posix.

- Elección de las funciones de Map y Reduce: indica si se permite al usuario final definir sus propias funciones de Map-Reduce.
- Software libre: indica si la librería tiene licencia abierta o, por el contrario es software propietario.

Como demuestra el cuadro 3.2, ambas soluciones analizadas son muy similares. Por tanto, se ha escogido utilizar la implementación de Mimir, como consecuencia principal de que no hace un uso abusivo operaciones de entrada y salida en disco, y por ello obtiene un mayor rendimiento.

3.3 Requisitos

En esta sección se especificarán cada uno de los requisitos que se han tenido en cuenta tanto a la hora de diseñar la interfaz sobre MPI-IO, como al modificar la librería de Mimir.
Para la obtención de todos los requisitos de software que se han tenido en cuenta durante la realización de este proyecto se ha seguido la especificación que hace la IEEE en su guía de buenas prácticas [30]. Según dicha guía, un requisito software es una capacidad o funcionalidad que debe tener un proyecto software, ya sean aquellas que necesita tener el usuario como aquellas que debe poseer la aplicación. Toda especificación de requisitos para estar hecha de una manera correcta tiene que seguir las siguientes características:

- Completa: debe describir todas las necesidades relevantes del sistema.
- Consistente: no debe contener conflictos entre los requisitos.
- Correcta: no debe contener errores.
- Modificable: se debe poder realizar cambios de una forma correcta.
- Verificable: debe existir un proceso para verificar cada uno de los requisitos.
- Trazable: el origen de cada uno de los requisitos se encuentra marcado de forma clara.
- No ambigua: cada uno de los requisitos únicamente debe tener una interpretación.

Para clasificar los requisitos, seguiremos la siguiente clasificación, que los divide funcionales y no funcionales.

1. Requisitos funcionales: detallan las funciones que tiene que cumplir el sistema.
2. No funcionales: indican las restricciones que se han impuesto en el sistema. A su vez las podemos subdividir en:
   (a) Consumo de recursos: indican los recursos máximos hardware (cpu, memoria) que puede utilizar el sistema.

(b) Rendimiento: Indican el tiempo máximo de respuesta y la velocidad de ejecución.

(c) Fiabilidad: indica el número de fallos que se permiten en el sistema.

(d) Manejo de errores: cómo el sistema debe manejar los fallos

(e) Requisitos de interfaz: Cómo el sistema se debe comunicar con otras aplicaciones, o con otros usuarios.

(f) Restricciones: limitaciones del sistema de cara a otros lenguajes de programación, arquitectura.

(g) Seguridad: detalla la seguridad que tiene que tener el sistema.

Durante la elaboración de este proyecto se han tenido en cuenta una gran cantidad de requisitos software, con el fin de facilitar su lectura y compresión se ha usado una tabla como 3.3

Identificador	Identificador del requisito. Una "I" en el identificador indica que es un requisito de la interfaz. Una "M" indica que se trata de un requisito de la librería de Mimir.
Título	Título del requisito.
Prioridad	Indica la importancia del requisito, que se puede encontrar en tres categorías (alta, media y baja).
Fuente	Identifica el origen del requisito. Esta puede ser cualquier persona involucrada en el proyecto o puede provenir por cómo está definido el estándar MPI.
Estabilidad	Indica si el requisito variará con el tiempo, en tal caso será una estabilidad baja o si por el contrario apenas será modificado dando lugar a una estabilidad alta.
Descripción	Describe el requisito de forma detallada.

Cuadro 3.3: *Tabla de ejemplos de los requisitos*

3.3.1 Requisitos funcionales

Identificador	RF-I-01.
Título	Conexión con Hadoop.
Prioridad	Alta.
Fuente	Tutor.
Estabilidad	Alta.
Descripción	La nueva interfaz deberá conectarse con el DataNode de Hadoop para tener acceso a los ficheros del mismo.

Cuadro 3.4: *Requisito funcional RF-I-01*

Identificador	RF-I-02.
Título	Apertura de un fichero.
Prioridad	Alta.
Fuente	Tutor.
Estabilidad	Alta.
Descripción	Se permitirá la apertura de un fichero de HDFS en una aplicación de MPI, tanto en el modo de lectura como en el modo de escritura.

Cuadro 3.5: *Requisito funcional RF-I-02*

Identificador	RF-I-03.
Título	Cierre de un fichero.
Prioridad	Alta.
Fuente	Tutor.
Estabilidad	Alta.
Descripción	El sistema permitirá el cierre de un fichero de HDFS abierto previamente en una aplicación de MPI.

Cuadro 3.6: *Requisito funcional RF-I-03*

Identificador	RF-I-04.
Título	Lectura con offset explícito.
Prioridad	Alta.
Fuente	Tuto.
Estabilidad	Alta.
Descripción	El sistema permitirá la lectura de un fichero usando un offset explícito, es decir, incándo el offset en la llamada de la función.

Cuadro 3.7: *Requisito funcional RF-I-04*

Identificador	RF-I-05.
Título	Lectura con offset implícito.
Prioridad	Alta.
Fuente	Tutor.
Estabilidad	Alta.
Descripción	El sistema permitirá la lectura de un fichero usando el puntero interno almacenado en el descriptor del fichero.

Cuadro 3.8: *Requisito funcional RF-I-05*

Identificador	RF-I-06.
Título	Lectura con puntero compartido.
Prioridad	Alta.
Fuente	Tutor.
Estabilidad	Alta.
Descripción	El sistema permitirá la lectura de un fichero por varios procesos, usando el puntero compartido que se encuentra almacenado dentro del descriptor del fichero.

Cuadro 3.9: *Requisito funcional RF-I-06*

Identificador	RF-I-07.
Título	Definición de vistas para lectura.
Prioridad	Alta.
Fuente	Tutor.
Estabilidad	Alta.
Descripción	El sistema permitirá la definición de vistas en un fichero que se encuentre abierto para lectura.

Cuadro 3.10: *Requisito funcional RF-I-07*

Identificador	RF-I-08.
Título	Escritura de un fichero.
Prioridad	Alta.
Fuente	Tutor.
Estabilidad	Alta.
Descripción	El sistema permitirá la escritura en un fichero HDFS, que se encuentre abierto para escritura.

Cuadro 3.11: *Requisito funcional RF-I-08*

Identificador	RF-I-09.
Título	Borrado de un fichero.
Prioridad	Alta.
Fuente	Tutor.
Estabilidad	Alta.
Descripción	El sistema permitirá el borrado de un fichero que se encuentre almacenado dentro de HDFS

Cuadro 3.12: *Requisito funcional RF-I-09*

Identificador	RF-I-10.
Título	Mover el puntero interno de lectura.
Prioridad	Alta.
Fuente	Tutor.
Estabilidad	Alta.
Descripción	El sistema permitirá mover el puntero interno de lectura de un fichero.

Cuadro 3.13: *Requisito funcional RF-I-10*

Identificador	RF-I-11.
Título	Consulta del tamaño de un fichero.
Prioridad	Alta.
Fuente	Tutor.
Estabilidad	Alta.
Descripción	El sistema permitirá la consulta del tamaño de un fichero.

Cuadro 3.14: *Requisito funcional RF-I-11*

Identificador	RF-M-12.
Título	Localidad.
Prioridad	Alta.
Fuente	Tutor.
Estabilidad	Alta.
Descripción	La librería solo asignará bloques a un proceso que se esté ejecutando en su misma máquina.

Cuadro 3.15: *Requisito funcional RF-M-12*

Identificador	RF-M-13.
Título	Definición de funciones de Map y Reduce.
Prioridad	Alta.
Fuente	Tutor.
Estabilidad	Alta.
Descripción	El sistema permitirá al usuario modificar las funciones de Map y Reduce.

Cuadro 3.16: *Requisito funcional RF-M-13*

3.3.2 Requisitos no funcionales

Identificador	RNF-I-01.
Título	Estandar MPI-IO.
Prioridad	Alta.
Fuente	Tutor.
Estabilidad	Alta.
Descripción	El sistema seguirá la interfaz descrita en la sección 2.2.2

Cuadro 3.17: *Requisito no funcional RNF-I-01*

Identificador	RNF-I-02.
Título	Manejo de errores.
Prioridad	Alta.
Fuente	Tutor.
Estabilidad	Alta.
Descripción	El sistema tratará los errores de la misma manera que son gestionados en MPICH.

Cuadro 3.18: *Requisito no funcional RNF-I-02*

Identificador	RNF-I-03.
Título	Semántica de HDFS.
Prioridad	Alta.
Fuente	Tutor.
Estabilidad	Alta.
Descripción	El sistema seguirá la semántica de HDFS permitiendo solo aquellas funciones que permita este sistema de ficheros.

Cuadro 3.19: *Requisito no funcional RNF-I-03*

Identificador	RNF-I-04.
Título	Indicar que el fichero que se desea abrir se encuentra en HDFS
Prioridad	Alta.
Fuente	Tutor.
Estabilidad	Alta.
Descripción	Para indicar que el sistema de ficheros se encuentra en HDFS, al llamar a la función de apertura se debe indicar el sistema de ficheros. En el caso de HDFS debe ser "hdfs" o "HDFS".

Cuadro 3.20: *Requisito no funcional RNF-I-04*

Identificador	RNF-I-05.
Título	Lenguaje de programación de la interfaz.
Prioridad	Alta.
Fuente	Tutor.
Estabilidad	Alta.
Descripción	La interfaz estará realizada en el lenguaje de programación C.

Cuadro 3.21: *Requisito no funcional RNF-I-05*

Identificador	RNF-M-06.
Título	Lenguaje de programación de la librería Mimir.
Prioridad	Alta.
Fuente	Tutor.
Estabilidad	Alta.
Descripción	Las modificaciones de Mimir estarán realizadas en lenguaje de programación C++.

Cuadro 3.22: *Requisito no funcional RNF-M-06*

3.4 Marco regulatorio

En esta sección se describirá la licencia bajo la que se distribuyen cada una de las opciones software utilizadas para la desarrollo de este TFG. En primer lugar, el framework de Hadoop se distribuye bajo una licencia de software libre, más en concreto bajo la licencia Apache [31] que fue creada específicamente por la Apache Software Foundation. Esta licencia permite la modificación y la redistribución del software, siempre y cuando se incluya la información final indicando que el código utilizado tenía licencia apache.

Por otro lado, la licencia bajo la que se distribuye MPICH es una licencia de software libre. Esta licencia permite la redistribución, la modificación y además la redistribución de la versión modificada.

Para terminar el último componente software usado, Mimir, se distribuye al igual que las an-

teriores, es decir, bajo una licencia de software libre. La licencia de este software permite la redistribución, la modificación y la redistribución del software modificado. Sin embargo en este caso, el software debe redistribuirse bajo la misma licencia que tiene actualmente.

Capítulo 4

Diseño

En este capítulo se desarrollan todas las ideas que se han llevado a cabo a la hora de diseñar este proyecto. En primer lugar se describen los componentes de la interfaz desarrollada (sección 4.1); por otro lado se describe cómo se ha modificado la librería de Mimir para conseguir un correcto funcionamiento con el sistema de ficheros de HDFS (sección 4.2.

4.1 Interfaz MPI-IO sobre HDFS

Durante esta sección, se describirá cómo se ha construido el método para añadir el sistema de ficheros de HDFS sobre la implementación de MPI-IO que realizó MPICH.
La interfaz desarrollada se ha incluido dentro de Romio, dentro de la interfaz abstracta ADIO. Estos conceptos fueron detallados en el capítulo 2 del presente documento. La siguiente figura, 4-1, muestra la integración del sistema de ficheros de HDFS dentro de MPI-IO.

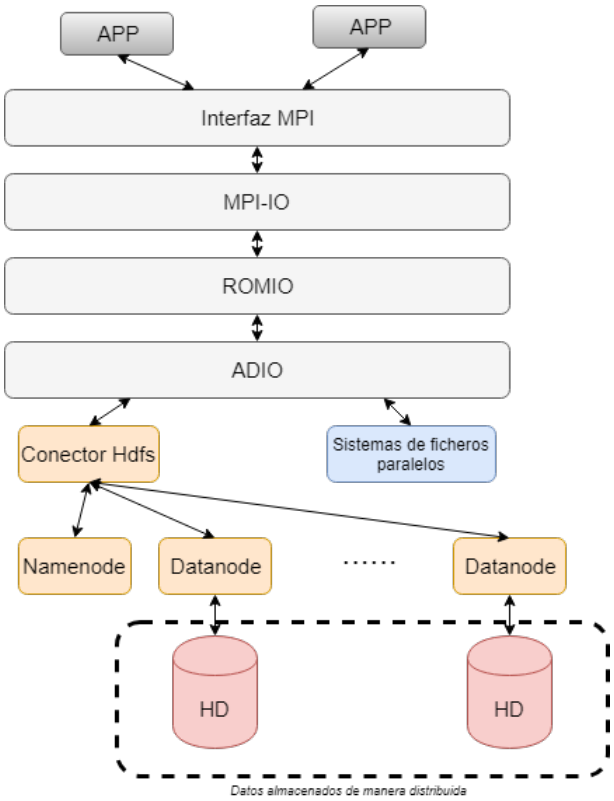

Figura 4-1: *Integración de HDFS en Romio*

La implementación que se va a realizar de ADIO, para el sistema de ficheros HDFS, va a ser una implementación que siga la semántica de HDFS. Por ello, todas las operaciones que HDFS no soporta no serán incluidas dentro de la implementación. Las principales limitaciones que tiene HDFS fueros detalladas anteriormente (capítulo 2). En las siguientes secciones se analizarán, aquellas funciones de MPI que se van a incluir en la implementación y aquellas que se excluirán, con el objetivo de facilitar su compresión, las funciones implementadas se clasificarán en tres conjuntos: primero, aquellas que permiten la manipulación de los ficheros (sección: 4.1.1), en segundo lugar aquellas que permiten definir vistas en los ficheros sección(4.1.2); y ,en tercer lugar, las funciones que permiten el acceso a los datos (sección 4.1.3).

4.1.1 Manipulación de los ficheros

En el desarrollo de esta sección se describirán las funciones que ofrece MPI-IO, y que permiten la manipulación de los ficheros.

MPI_File_open

Esta función permite la apertura de un fichero con el objetivo de que se permita su lectura o escritura en el mismo. MPI define muchos modos de apertura de un fichero, pero HDFS

solo permite tres modos debido a su funcionamiento, por ello solo se dará soporte a estos tres modos:

- *MPI_MODE_RDONLY*: permite la apertura de un fichero para leer su contenido. Esta función puede ser ejecutada por varios procesos a la vez, por lo que en el comunicador puede haber más de un proceso.
- *MPI_MODE_WRONLY*: HDFS solo permite un proceso escribiendo a la vez a un fichero. A causa de esto, esta llamada debe ser conovcada únicamente por un proceso, es decir, el proceso que la llama debe ser el único que abra el fichero. Además, es importante destacar que cuando se abre un fichero en el modo de escritura, si este fichero ya existía previamente en el sistema, será truncado a cero.
- *MPI_MODE_APPEND*: al igual que el modo de apertura anterior, la función de apertura de un fichero en este modo solo se puede ejecutar con un proceso. A diferencia del modo de apertura anterior en este modo, HDFS permite la escritura al final de un fichero, pero en ningún caso se permite modificar su contenido.

MPI_File_close

Esta función permite el cierre de un fichero. Esta función tiene que ser ejecutada por todos los procesos que han abierto el archivo. En el caso de que el fichero esté abierto en el modo de escritura y se esté escribiendo en él, los datos no se encontrarán disponibles hasta que se cierre el fichero.
Con el objetivo de permitir que las aplicaciones puedan leer los datos antes del cierre del fichero, se va a implementar un hint en el cual será el usuario el encargado de decidir cuando los bytes de un fichero se encuentran disponibles. Para ello el usuario tendrá que incluir en un objeto MPI_Info el atributo write_mode, con el valor de “hflush’.

MPI_File_delete

Esta función elimina un fichero dentro de HDFS.

MPI_File_set_size

Esta función permite cambiar el tamaño de un fichero. Por el contrario, debido a que HDFS no permite la modificación de los datos ya escritos en un fichero.

MPI_File_get_size

Esta función devuelve el tamaño de un fichero de HDFS.

MPI_File_get_amode

Devuelve el modo de apertura de un fichero.

MPI_File_set_info

Permite definir información sobre un fichero abierto. El objeto *MPI_info* sirve para definir algunos parámetros sobre cómo se produce el acceso a un fichero. Estos objetos almacenan información del tipo <*atributo,valor*>. MPI tiene reservados una gran cantidad de atributos, estos se denominan hint y sirven para indicar algunas opciones de acceso a los ficheros [6]. En la implementación realizada se han decidido reservar cuatro valores, con el objetivo de permitir a los usuarios finales de la interfaz, la elección de ciertos parámetros de configuración. Estos valores son:

- write_mode: permite seleccionar cuando los datos escritos en un fichero se encuentran disponibles para el usuario. Por defecto, y a no ser que el usuario defina el valor "HFLUSH", los datos se encontrarán disponibles tras el cierres del fichero.
- hdfs_buffersize: permite la elección del tamaño del buffer interno de HDFS.
- hdfs_replication: permite la elección del número de réplicas que se harán del fichero creado. solo tiene sentido cuando se crea un fichero.
- hdfs_blocksize: permite la elección del tamaño del bloque con el cual será almacenado un fichero. Solo tiene sentido cuando se crea un fichero.

MPI_File_get_info

Devuelve un objeto *MPI_info* asociado a un fichero. En este objeto *MPI_Info* se encuentran almacenados todos los atributos y valores definidos para un fichero.

4.1.2 Vistas de los ficheros

Las vistas de MPI definen las partes del fichero que se encuentran disponibles para cada uno de los procesos, de este modo, se pueden configurar que cada uno de los ficheros accedan a partes no contiguas del fichero. Las funciones que permiten la manipulación de las vistas son:

MPI_File_set_view

Permite definir una nueva vista para un fichero. En esta implementación de la interfaz, se permitirá definir una nueva vista a un fichero que se encuentre abierto en el modo de lectura. Por el contrario, en el modo de escritura no se podrán definir las vistas, porque se pueden crear

vistas que modifiquen la parte ya escrita de un fichero, y HDFS no permite la modificación de los datos.

MPI_File_get_view

Esta función permite obtener la vista que se ha definido para un fichero.

4.1.3 Acceso a los datos de los ficheros

La interfaz de MPI-IO define una gran cantidad de funciones que permiten el acceso a los datos (sección: 2.2.2). Durante el apartado actual se proporcionará una visión global de aquellas funciones que han sido implementadas y se describen cuales han sido sus limitaciones, en el caso de que las posean. Con el objetivo de realizar una clasificación, se han dividido las diferentes funciones en relación con posicionamiento dentro del fichero. En primer lugar, se verán las funciones con puntero explícito; en segundo lugar, las funciones con puntero individual; y para terminar, aquellas funciones que utilizan el puntero compartido.
Para mostrar de una forma más comprensible qué funciones están implementadas y cuales no, se ha creado la siguiente tabla [4.29].

Nombre función
Descripción
Soportada
Causas no implementación
Observaciones

Cuadro 4.1: *Ejemplo de tabla de descripción del diseño de las funciones*

Donde cada uno de los campos se corresponde con:

- Función: indica el nombre de la función de MPI.
- Descripción: detalla la funcionalidad que tiene esa función.
- Soportada: indica si la implementación va a soportar esta función o, si por el contrario, no lo hará.
- Causas para no implementación: en el caso de que la función no haya sido implementada, indica las causas por las cuales no lo fue.
- Observaciones: indica otras características del diseño.

Puntero individual

A continuación se presentan las funciones que hacen uso del puntero individual de cada fichero.

Nombre función	MPI_File_read_at
Descripción	Permite la lectura de un fichero indicando un offset explícito.
Soportada	Si.
Causas no implementación	-
Observaciones	-

Cuadro 4.2: *Función MPI_File_read_at*

Nombre función	MPI_File_write_at
Descripción	Escribe en un fichero con un offset indicado por argumento.
Soportada	No.
Causas no implementación	HDFS no permite la modificación de los bytes ya escritos en un fichero.
Observaciones	-

Cuadro 4.3: *Función MPI_File_write_at*

Nombre función	MPI_File_iread_at
Descripción	Versión no bloqueante que permite la lectura de un fichero con un offset indicado como argumento.
Soportada	No.
Causas no implementación	HDFS no dispone de llamadas al sistema que sean no bloqueantes.
Observaciones	Se puede usar pero se comporta de manera similar a la llamada bloqueante (cuadro 4.2).

Cuadro 4.4: *Función MPI_File_iread_at*

Nombre función	MPI_File_iwrite_at
Descripción	Escribe en un fichero con un offset indicado como argumento de un modo no bloqueante.
Soportada	No.
Causas no implementación	HDFS no permite la modificación de los bytes ya escritos en un fichero.
Observaciones	-

Cuadro 4.5: *Función MPI_File_iwrite_at*

Nombre función	MPI_File_read_at_all
Descripción	Versión colectiva de lectura con offset explícito.
Soportada	Si.
Causas no implementación	-
Observaciones	-

Cuadro 4.6: *Función: MPI_File_read_at_all*

Nombre función	MPI_File_write_at_all
Descripción	Versión de escritura colectiva con un offset explícito.
Soportada	No.
Causas no implementación	HDFS no permite la escritura en un fichero con un offset explícito, puesto que no permite la modificación de los bytes ya escritos en un fichero.
Observaciones	-

Cuadro 4.7: *Función: MPI_File_write_at_all*

Nombre función	MPI_File_iread_at_all
Descripción	Versión no bloqueante de lectura con un offset explícito.
Soportada	No.
Causas no implementación	HDFS no incluye funciones de lectura no bloqueantes.
Observaciones	Se puede usar pero, tiene un comportamiento similar que la llamada bloqueante (cuadro 4.6).

Cuadro 4.8: *Función: MPI_File_iread_at_all*

Nombre función	MPI_File_iwrite_at_all
Descripción	Versión colectiva no bloqueante de escritura con un offset explícito
Soportada	No.
Causas no implementación	HDFS no dispone de funciones de escritura no bloqueantes. Además, no se puede escribir con un offset explícito.
Observaciones	-

Cuadro 4.9: *Función: MPI_File_iwrite_at_all*

Nombre función	MPI_File_read_at_all_begin MPI_File_read_at_all_end
Descripción	Función de lectura colectiva no bloqueante con un offset explícito, la cual se encuentra divida en dos funciones, una que inicia la operación de lectura y otra que la finaliza.
Soportada	No.
Causas no implementación	HDFS no dispone de funciones de lectura no bloqueantes.
Observaciones	Se puede usar pero, tiene un comportamiento similar que la llamada bloqueante (cuadro 4.6).

Cuadro 4.10: *función de lectura no bloqueante y dividida en dos funciones con offset explícito*

Nombre función	MPI_File_write_at_all_begin MPI_File_write_at_all_end
Descripción	Función de escritura colectiva no bloqueante con un offset explícito la cual se encuentra dividida en dos funciones, la primera inicia la lectura y la segunda función la devuelve.
Soportada	No.
Causas no implementación	HDFS no dispone de funciones no bloqueantes. Además la escritura solo se puede realizar por un proceso, y tampoco se puede escribir en HDFS con un offset explícito.
Observaciones	-

Cuadro 4.11: *Función de escritura no bloqueante dividida en dos funciones con offset explícito.*

Offset implícito

Nombre función	MPI_File_read
Descripción	Permite la lectura secuencial en un fichero con el puntero de posicionamiento interno.
Soportada	Si.
Causas no implementación	-
Observaciones	-

Cuadro 4.12: *Función: MPI_File_read*

Nombre función	MPI_File_write
Descripción	Permite la escritura secuencial de un fichero usando el puntero interno del archivo.
Soportada	Si.
Causas no implementación	-
Observaciones	Esta función esta implementada pero, solo puede tener un proceso el fichero abierto en el modo de escritura.

Cuadro 4.13: *Función: MPI_File_write*

Nombre función	MPI_File_iread
Descripción	Versión no bloqueante de lectura usando el puntero interno del archivo.
Soportada	No.
Causas no implementación	HDFS no dispone de las funciones de lectura no bloqueantes.
Observaciones	Se puede usar pero, tiene un comportamiento similar que la llamada bloqueante (cuadro 4.12).

Cuadro 4.14: *Función: MPI_File_iread*

Nombre función	MPI_File_iwrite
Descripción	Versión no bloqueante de escritura usando el puntero interno del archivo.
Soportada	No.
Causas no implementación	HDFS no dispone de funciones de escritura no bloqueantes.
Observaciones	Se puede usar pero, tiene un comportamiento similar que la llamada bloqueante (cuadro 4.13).

Cuadro 4.15: *Función: MPI_File_iwrite*

Nombre función	MPI_File_read_all
Descripción	Versión colectiva en la cual todos los procesos leen el fichero usando el puntero interno del fichero.
Soportada	Si.
Causas no implementación	-
Observaciones	-

Cuadro 4.16: *Función: MPI_File_read_all*

Nombre función	MPI_File_write_all
Descripción	Función colectiva de escritura en la cual todos los procesos escriben en la misma parte del fichero.
Soportada	No.
Causas no implementación	HDFS solo permite un proceso escribiendo en el mismo fichero de manera simultánea.
Observaciones	Se puede usar pero, tiene un comportamiento similar que la llamada no colectiva (cuadro 4.13).

Cuadro 4.17: *Función: MPI_File_write_all*

Nombre función	MPI_File_iread_all
Descripción	Versión no bloqueante colectiva de lectura de un fichero usando el puntero interno del fichero.
Soportada	No.
Causas no implementación	HDFS no tiene funciones no bloqueantes de lectura.
Observaciones	Se puede usar pero, tiene un comportamiento similar que la llamada bloqueante (cuadro 4.16).

Cuadro 4.18: *Función: MPI_File_read_all*

Nombre función	MPI_File_iwrite_all
Descripción	Función de escritura colectiva no bloqueante que permite la escritura haciendo uso del puntero interno del fichero.
Soportada	No.
Causas no implementación	HDFS no dispone de función para las escrituras no bloqueantes.
Observaciones	-

Cuadro 4.19: *Función: MPI_File_iwrite_all*

Nombre función	MPI_File_read_all_begin MPI_File_read_all_end
Descripción	Función de lectura no bloqueante, colectiva y dividida en dos funciones.
Soportada	No.
Causas no implementación	HDFS no dispone de funciones de lectura no bloqueantes.
Observaciones	Se puede usar pero, tiene un comportamiento similar que la llamada bloqueante (cuadro 4.16).

Cuadro 4.20: *Función de lectura no bloqueante dividida usando puntero interno del fichero*

Nombre función	MPI_File_write_all_begin MPI_File_write_all_end
Descripción	Función de escritura no bloqueante y colectiva dividida en dos funciones.
Soportada	No.
Causas no implementación	HDFS no dispone de funciones de escritura no bloqueantes. Además no se puede escribir en un fichero con más de un proceso en HDFS
Observaciones	-

Cuadro 4.21: *Función de escritura no bloqueante dividida usando el puntero interno del fichero*

Puntero compartido

Nombre función	MPI_File_read_shared
Descripción	Función de lectura en la que los procesos usan el puntero compartido del fichero.
Soportada	Si.
Causas no implementación	-
Observaciones	-

Cuadro 4.22: *Función: MPI_File_read_shared*

Nombre función	MPI_File_write_shared
Descripción	Función de escritura en la cual los procesos utilizan el puntero compartido para la escritura del fichero.
Soportada	No.
Causas no implementación	HDFS no permite que más de un proceso escriba en el mismo fichero de manera simultánea.
Observaciones	-

Cuadro 4.23: *Función: MPI_File_write_shared*

Nombre función	MPI_File_iread_shared
Descripción	Función no bloqueante de lectura en la que los procesos utilizan el puntero compartido para la lectura del fichero.
Soportada	No.
Causas no implementación	HDFS no permite llamadas no bloqueantes.
Observaciones	Se puede usar pero, tiene un comportamiento similar que la llamada bloqueante (cuadro 4.24).

Cuadro 4.24: *Función: MPI_File_iread_shared*

Nombre función	MPI_File_iwrite_shared
Descripción	Función no bloqueante de escritura, en la cual los procesos utilizan el puntero compartido para la escritura en el fichero.
Soportada	No.
Causas no implementación	HDFS no define operaciones de escritura no bloqueantes, y además no permite a más de un proceso escribiendo en el mismo fichero simultáneamente.
Observaciones	-

Cuadro 4.25: *Función: MPI_File_iwrite_shared*

Nombre función	MPI_File_read_ordered
Descripción	Función colectiva de lectura, usando el puntero compartido dentro de cada uno de los punteros.
Soportada	Si
Causas no implementación	-
Observaciones	-

Cuadro 4.26: *Función: MPI_File_read_ordered*

Nombre función	MPI_File_write_ordered
Descripción	Función de escritura colectiva en un fichero.
Soportada	No.
Causas no implementación	HDFS no permite la escritura de forma simultánea en el mismo fichero por más de un proceso.
Observaciones	-

Cuadro 4.27: *Función: MPI_File_write_ordered*

Nombre función	MPI_File_read_ordered_begin MPI_File_read_ordered_end
Descripción	Función de lectura compartida de un fichero usando el puntero compartido interno del mismo.
Soportada	No.
Causas no implementación	HDFS no dispone de llamadas al sistema no bloqueantes.
Observaciones	Se puede usar pero, tiene un comportamiento similar que la llamada bloqueante. (cuadro 4.26).

Cuadro 4.28: *Función de lectura no bloqueante dividida usando el puntero compartido del fichero*

Nombre función	MPI_File_write_ordered_begin MPI_File_write_ordered_end
Descripción	Funciones de lectura colectiva con el puntero compartido y no bloqueantes, con la con el punte
Soportada	No.
Causas no implementación	HDFS no permite la escritura simultanea por más de un proceso en el mismo fichero.
Observaciones	-

Cuadro 4.29: *Función de escritura no bloqueante dividida usando el puntero compartido del fichero*

Para finalizar se ofrece una lista con las funciones de acceso a los ficheros que sí fueron implementadas.

- MPI_File_read_at (Cuadro: 4.2).
- MPI_File_read_at_all (Cuadro: 4.6).
- MPI_File_read (Cuadro: 4.12).
- MPI_File_write (Cuadro: 4.13).
- MPI_File_read_all (Cuadro: 4.16).
- MPI_File_read_shared (Cuadro: 4.24).
- MPI_File_read_ordered (Cuadro: 4.26).

4.2 Librería de Map-Reduce

En esta sección, se va a detallar el funcionamiento que presenta inicialmente la librería de Mimir con el objetivo de poder adaptarla posteriormente a nuestro funcionamiento. Para ello nos centraremos en tres aspectos fundamentalmente:

- Cómo abre los ficheros.
- Cómo parte el fichero entre los diferentes procesos.
- Cómo soluciona el problema en las fronteras de los bloques para no leer dos veces las mismas palabras.

4.2.1 Apertura de ficheros en Mimir

La librería de Mimir solo funciona con sistema de ficheros Posix, esto se debe a que no utiliza la interfaz de MPI para acceder a los ficheros.
Cuando a un programa escrito usando la librería de Mimir se le pasa un fichero, en primer lugar, el proceso cero consulta el tamaño del fichero y se lo manda al resto de procesos. Cuando

éstos lo han recibido, proceden a calcular de forma conjunto los fragmentos del fichero que les corresponde a cada uno de los procesos.

4.2.2 Particionado del fichero entre los diferentes procesos.

Una vez que el proceso cero envía el tamaño del fichero a cada uno de los procesos, estos calculan qué porción del fichero les toca procesar a cada uno. Para ello:

1. Dividen el fichero en bloques de 64 MB.

2. Calculan cuántas porciones de 64 MB del fichero les toca a cada uno de los procesos.

3. Cada uno de los procesos almacena los pedazos del fichero que le toca procesar, sabiendo el byte en el cual deben comenzar la lectura y en cual debe finalizar.

De este modo, los procesos siempre procesan partes contiguas del fichero, pero esos datos no tienen por que estar en la misma máquina en la cual se está ejecutando el proceso y por lo tanto que se tengan que enviar por red, como se muestra en la figura 4-2

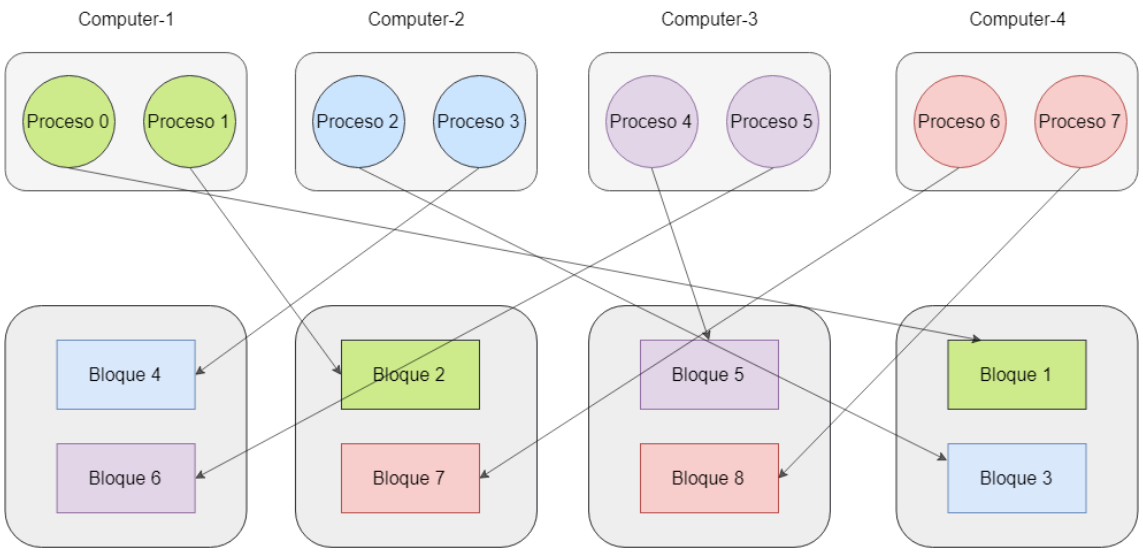

Figura 4-2: *Reparto del fichero en mimir sin localidad*

Para solucionar este problema se propone la siguiente solución: el particionado de los bloques se hace por máquina, no por proceso y en cada una de las máquinas solo se procesan los bloques que se encuentran almacenados en ella, de este modo no es necesario enviar el bloque completo por la red. Una vez que se han asignado los bloques, los procesos de cada una de las máquinas acceden a los bloques del fichero de forma paralela (figura 4-3).

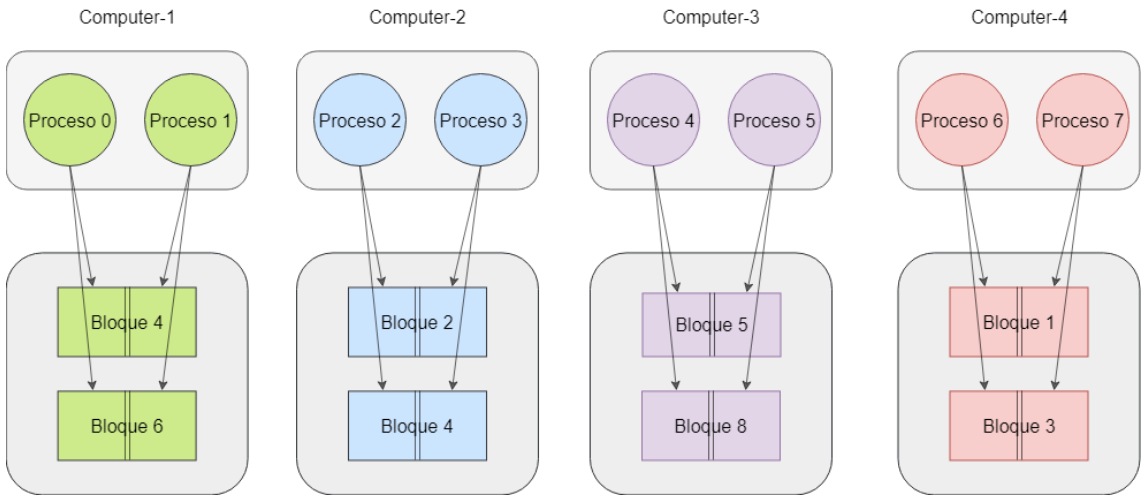

Figura 4-3: *Reparto del fichero en mimir con localidad*

4.2.3 Solución del conflicto en la frontera

El problema de la frontera resulta porque los procesos leen por bytes pero, al tener que procesar palabras enteras o líneas, por lo tanto no perciben hasta donde tienen que leer. Por ejemplo supongamos, que tenemos el siguiente fragmento de un fichero (figura 4-4).

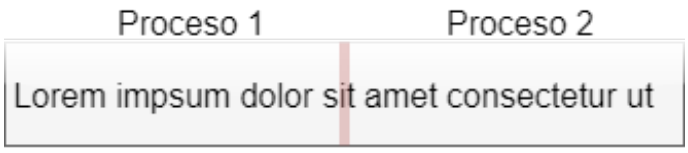

Figura 4-4: *Ejemplo del problema en la frontera*

Como se observa, el proceso 0 leerá : *Lorem impsum dolor si*; mientras que el proceso 1 leerá: *t aemet consecteur ut*; por lo tanto ninguno de los dos leerá la palabra *sit* correctamente. Para solucionar este problema Mimir envía los datos de un proceso al siguiente, desde el último salto de línea. De este modo conseguimos que siempre sea el proceso siguiente el que procesa los datos de la frontera. Como consecuencia de esto, para el correcto funcionamiento de Mimir, hay que definir un tamaño máximo de línea que será la parte que un programa de Mimir podrá intercambiar con otros procesos.

Capítulo 5

Implementación y desarrollo

En este capítulo, se detalla toda la implementación que se ha llevado a cabo en el sistema. En primer lugar se da una explicación detallada sobre cómo se han implementado las funciones de la interfaz de MPI-IO con el objetivo de que esta de soporte al sistema de ficheros de HDFS (sección 5.1). En segundo lugar, se describe como se han implementado las modificaciones necesarias para que Mimir pueda realizar operaciones de Map-Reduce sobre los ficheros almacenados en HDFS (sección 5.2).

5.1 Implementación de la interfaz

La implementación de la interfaz se ha hecho en el lenguaje de programación C; debido a que Hadoop se encuentra escrito en java ha sido necesario el uso de librerías auxiliares para poder crear la interfaz. En concreto, la librería utilizada es libhdfs [32], la cual hace uso de Interfaz nativa de java (JNI) [33], esta librería a su vez hace uso de la librería libjvm, la cual utiliza la máquina virtual de java.
La figura 5-1 representa la iteración entre las diferentes librerías que usa nuestra implementación.

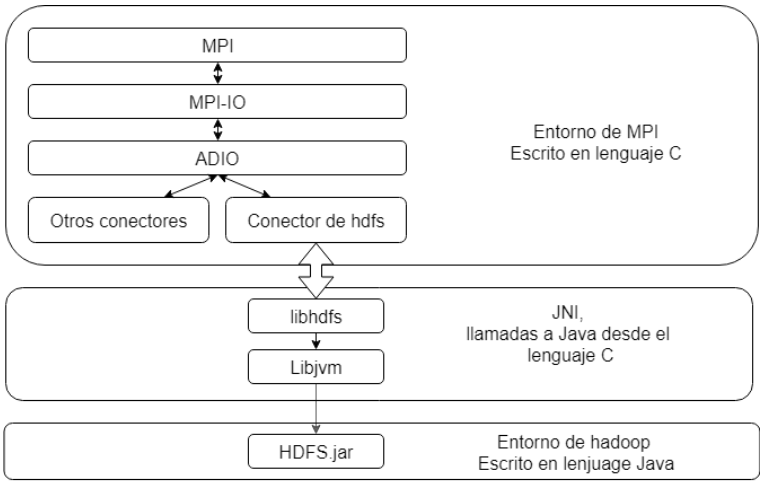

Figura 5-1: *Interacción entre las diferentes librerías*

5.1.1 Estructura de la aplicación

Con el objetivo de realizar una implementación organizada, este ha sido el árbol de directorios que se ha utilizado para la implementación.

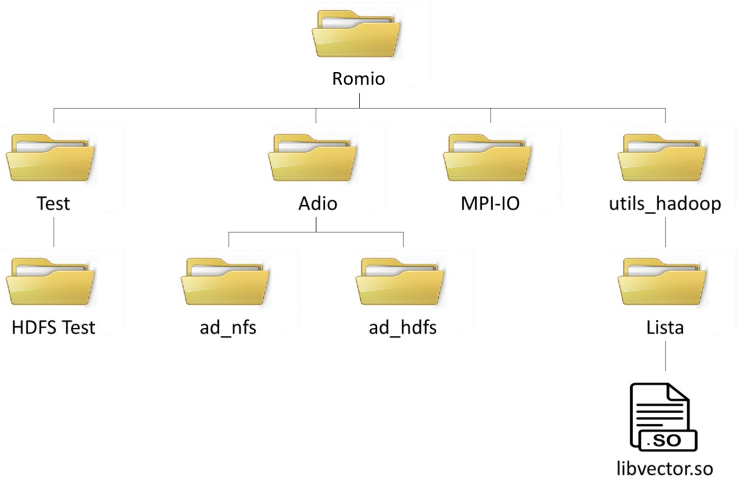

Figura 5-2: *Estructura de carpetas de la implementación de la interfaz*

En la figura se encuentran 4 carpetas que son importantes y donde está el grueso de nuestra implementación:

- HDFS test: Dentro de esta carpeta se encuentran aplicaciones de prueba para ejecutar la interfaz de HDFS sobre MPI.
- ad_hdfs: Dentro de esta carpeta se encuentra la implementación de las funciones de MPI

usando el sistema de ficheros de HDFS.

- mpi-io: En esta carpeta se encuentra la implementación de las funciones básicas de la interfaz MPI-IO.
- utils_hadoop: Dentro de esta carpeta se encuentran los ficheros necesarios para la compilación de la librería de la lista de punteros compartidos, necesaria para HDFS.

5.1.2 Modificaciones al iniciar MPI

Con el objetivo de implementar la interfaz se ha tenido que modificar el código de MPI, en concreto el de la función MPI_Init que sirve para iniciar una aplicación MPI. Para poder usar aplicaciones de MPI, que funcionen con la interfaz desarrollada es necesario el uso de la función de MPI_Init_thread, puesto que esta implementación, para compartir el puntero hace uso de un thread interno. Este thread se crea en el proceso 0 al iniciar la aplicación (MPI_Init_thread), y se destruye al finalizar (MPI_Finalize).

Conexión con el namenode de HDFS

En primer lugar, para poder trabajar con el sistema de ficheros de HDFS, cada uno de los procesos tiene que establecer una conexión con el NameNode. Esta conexión se realiza cuando se inicia el programa de MPI, es decir, cuando cada uno de los procesos llama a *MPI_Init_thread*. El usuario de la aplicación puede elegir en que máquina se encuentra el NameNode así como en que puerto se localiza, por medio de dos variables de entorno.
Las variables de entorno que el usuario debe configurar son:

- HADOOP_DEFAULT: En esta variable de entorno se define el nombre de la máquina en la cual se está ejecutando el NameNode.
- HADOOP_PORT: En esta variable de entorno se define el puerto en el que está escuchando el NameNode.

En el caso de que la aplicación de MPI se esté ejecutando en una máquina en la cual se encuentra instalada una instalación de Hadoop, se puede utilizar HADOOP_DEFAULT=default y HADOOP_PORT=0. De este modo, la aplicación cogerá la información de los archivos de configuración de Hadoop. Cuando el programa de MPI llama a *MPI_Finalize*, se produce la desconexión de cada uno de los procesos del del NameNode.

5.1.3 Función de apertura de un fichero

Como se ha indicado en la anteriormente (sección 4). La función de apertura solo permite tres modos de apertura.

- MPI_MODE_WRONLY: Modo de solo escritura.
- MPI_MODE_RDONLY: Modo de solo lectura.
- MPI_MODE_APPEND: Modo de escritura al final del fichero.

Durante la implementación de esta función se ha seguido la semántica de HDFS, esto quiere decir que en el caso de que se quiera crear un fichero se debe abrir el fichero en el modo de escritura. Si se abre un fichero en el modo escritura, y este ya existía, HDFS le elimina y le vuelve a crear con un tamaño de cero bytes, por lo tanto, toda la información que contuviese el fichero se ha perdido. En el caso de que el usuario haya definido los hints *hdfs_buf_size, hdfs_replication, hdfs_blocksize* (sección 4.1.1), en un objeto MPI_info, estos serán usados en la apertura del fichero. En el caso de que el usuario no los haya definido, se usarán los valores que tenga configurados la instalación de Hadoop.

5.1.4 Funciones de lectura

Durante esta sección, se detalla la implementación de las funciones de lectura. Como se detalló anteriormente hay tres tipos de funciones de lectura diferentes según el puntero que utilicen.

Offset explícito

Esta función lee de un fichero a partir de una posición indicada en la llamada a la función, esta posición se denomina offset, puesto que el offset es un argumento de esta función, no se tiene que actualizar la posición interna del puntero de lectura.

Puntero individual

Para la implementación del puntero individual, dentro del descriptor de un fichero MPI, se tiene almacenado la posición de lectura del proceso en el fichero. Cuando un proceso realiza una llamada que usa el puntero individual, se procede a actualizar la posición de este puntero. Este puntero también se puede modificar por medio de la función MPI_File_seek. En el caso de que se haya llegado al fin del fichero la función de lectura sigue ejecutando pero no devuelve nada.
Para conocer el número de bytes leídos, y con ello, saber cuando se ha finalizado de leer el fichero, hay que usar la función de MPI MPI_Get_count.

Puntero compartido

Para la implementación del puntero compartido, se necesitaba un proceso que fuera el encargado de gestionar los punteros compartidos para cada uno de los ficheros abiertos. El proceso seleccionado para ello ha sido el proceso 0. Este proceso cuando la aplicación llama a

MPI_Init_thread, crea un thread, y este a su vez crea una lista simple enlazada, en la cual, se almacenan los ficheros abiertos y su puntero compartido.
Cuando un proceso quiere acceder al puntero compartido de un fichero, le envía al proceso 0 una petición, y este es el encargado por medio del thread, de enviarle la información del puntero compartido. Cuando la aplicación llama a MPI_Finalize es cuando se mata al thread.

5.1.5 Vistas

Cuando se define una vista para un fichero, cada uno de los procesos solo leerá la parte del fichero que le corresponda según la vista que tiene establecida. De este modo se permite la lectura paralela de un fichero por varios procesos usando un puntero individual, ya que cada uno de los procesos solo lee la parte correspondiente con la vista que tenga definida.
En la construcción de esta interfaz, se ha implementado la optimización *data sieving*, es decir, cuando se lee de un fichero con una vista definida, se lee un trozo grande de fichero, y luego se le devuelve al proceso solo aquellos bytes que le corresponden de acuerdo a su vista. Esto es mucho más eficiente que realizar muchas peticiones de lectura pequeñas sobre un fichero.

5.1.6 Funciones de escritura

En esta sección se describen cómo se han implementado las funciones de escritura, puesto que la función de escritura en HDFS siempre escribe en el final del fichero, es un error, si a esta función se le pasa un offset como argumento, de este modo, las llamadas con un puntero explícito o compartido dan fallo.
Puesto que HDFS no permite que un fichero abierto para lectura, sea abierto para escritura, cuando un proceso desea abrir el fichero para escritura, envía el nombre al proceso 0, y este comprueba si está abierto, (Para ello comprueba si se encuentra en la lista de punteros compartidos). Por el contrario si que se puede leer un fichero en el cual se está escribiendo, siempre que la apertura en modo escritura se realice antes que la apertura de lectura, o bien cuando un proceso abra el fichero indicando el modo de apertura *MPI_MODE_APPEND*, es por ello, que se ha proporcionado al usuario un hint para indicar cuando los datos en el fichero se encuentran disponibles para su lectura. Para ello debe definir en un objeto MPI_Info la clave *write_mode* y el valor *hflush*, en el caso de que el usuario no defina este valor los datos que el proceso esté escribiendo en el fichero no estarán disponibles hasta el cierre cierre el fichero.w

5.2 Implementación de Mimir

Durante esta sección se describirán los principales cambios que se han hecho en la librería de Mimir. Los cambios realizados principalmente han sido tres:

- Modificación de la función de apertura de los ficheros.
- Modificación de la forma de reparto de los bloques de los ficheros.
- Modificación de la forma en la que los procesos procesan la zona de la frontera del bloque.

5.2.1 Función de apertura de los ficheros

Como se ha comentado en el capítulo 3 (Análisis), la librería de Mimir solo funciona con ficheros que siguen la semántica Posix. Puesto que HDFS no sigue esta semántica es necesario modificar la forma de apertura de los ficheros. Para ello en vez del uso de funciones Posix se ha modificado la librería para que solo utilice funciones de MPI. De este modo ya se permite la apertura de un fichero almacenado en HDFS.

5.2.2 Reparto de los bloques para cada fichero

Con el objetivo de incluir la localidad en la librería de Mimir se ha cambiado la función de particionado de los ficheros. De este modo, cada uno de los procesos solo procesan fragmentos que se encuentran almacenados de su máquina.

El siguiente pseudocódigo muestra cómo se realiza el particionado del fichero entre los diferentes procesos:

```
0   Funcion particionadoHDFS () {
1       Cada proceso envia el nombre de la maquina donde ejecuta al proceso 0
2       if (soy el proceso 0){
3           Obtengo la lista de bloques del fichero.
4           int numeroBloques = tamano del fichero / tamano del bloque
5       }
6       int bloquesAsignados = 0, procesoActual = 0, bloqueActual = 0
7       While(bloquesAsignados < numeroBloques){
8           if(bloqueActual almacenado en la maquina donde se ejecuta el proceso actual){
9               bloqueActual asignado al proceso actual
10              bloqueActual ++
11              bloquesAsignados++
12              procesoActual ++
13              if (procesoActual >= NumeroProcesosEn Ejecucion)
14                  procesoActual = 0
15          }
16          procesoActual ++ //Pasamos al siguiente proceso
17          if (procesoActual >= NumeroProcesosEnEjecucion)
18              procesoActual = 0 //Se vuelve al primer proceso
19      }
```

```
20          Mensaje de Broadcast del proceso 0 al resto de procesos. //envio de ficheros asignados a maquinas
21          Return
22      }
```

5.2.3 Problema en la frontera de bloque

Como se definió en el capítulo 3 (sección 4.2.3), cada uno de los procesos tiene que leer y procesar su porción del fichero que le fue asignada. Mimir va devolviendo en la función de Map líneas completas del fichero. Pero las particiones no tienen por que acabar en un final de línea, si no que por normal general acabarán en medio de esta. Con el objetivo de solucionar este problema de la forma más eficiente posible, se ha realizado una implementación en la que, cada uno de los procesos leen el bloque que les corresponde entero y 2000 bytes del siguiente bloque. De este modo cada proceso procesa su bloque entero y hasta el primer salto de línea del siguiente bloque. Con el objetivo de que no se procese dos veces la misma línea, cada uno de los procesos, descarta la primera línea de su bloque, ya que, esta fue procesada por el proceso anterior, a no ser que se trate del primer bloque del fichero, en cuyo caso, el proceso si que la procesará. En la anterior figura 5-3, las líneas rojas identifican donde acaban los bloques de

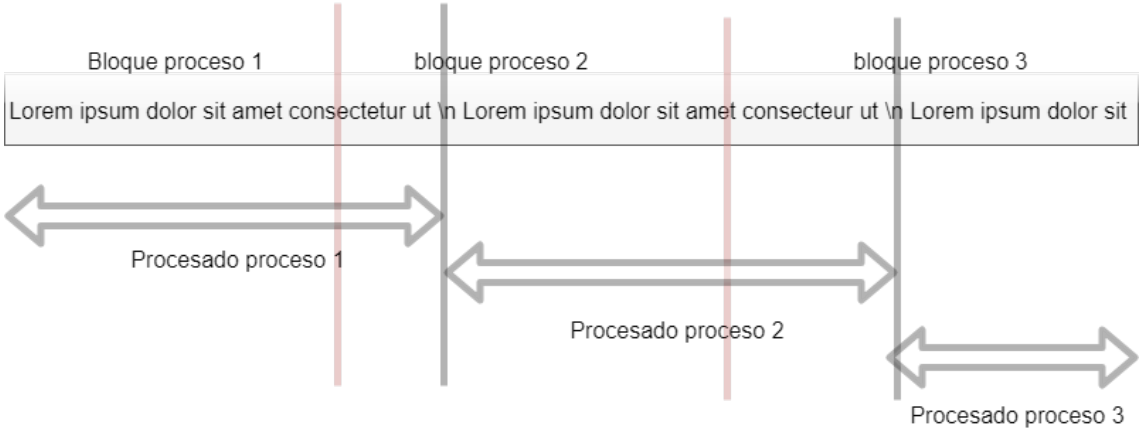

Figura 5-3: *Porción de fichero procesada por cada proceso*

cada uno de los procesos y, las líneas grises identifican que porción del fichero es la que procesa cada uno de los procesos.
Este proceso es similar al proceso utilizado por Hadoop dentro de su Map-Reduce.

Capítulo 6

Verificación Validación y evaluación

Este capítulo detalla la verificación, la validación y la evaluación de todo el proyecto. En primer lugar (sección 6.1), se realizará una verificación del funcionamiento del sistema, y se comprobará el cumplimiento de todos los requisitos de usuario, definidos en el capítulo de análisis 3. En segundo lugar (sección 6.2), se comprobará el rendimiento de la interfaz para MPI, realizando dos estudios de rendimiento.

- Comparativa del uso de la interfaz nativa de java contra el uso de MPI.
- Uso de la librería de Mimir con el objetivo de analizar cómo afecta la localidad implementada en dicha librería en el tiempo de acceso a los datos.

6.1 Verificación y validación

Para la comprobación y verificación del sistema creado, se van a proceder a realizar dos tipos de pruebas. En primer lugar, se realizarán pruebas de aceptación del sistema (seccion 6.1.1); posteriormente en segundo lugar, se harán pruebas de verificación (sección 6.1.2).

6.1.1 Pruebas de aceptación

Las pruebas de aceptación sirven para comprobar que el sistema funciona correctamente y cumple con todos los requisitos definidos en el capítulo 3 (sección 3.3).
Para realizar estos test, se han utilizado unas tablas como las mostradas a continuación:

Identificador	Sirve para identificar de forma unívoca la prueba.
RF relacionado	Cada uno de los requisitos que se han verificado con esta prueba.
Descripción	Describe el objetivo de la prueba.
Precondiciones	Indican las condiciones iniciales del sistema.
Procedimiento	Indican los pasos que deben seguirse para replicar la prueba.
Postcondiciones	Indican las condiciones finales del sitema tras la prueba.
Éxito	Indica si el sistema a pasado la prueba satisfactoriamente.

Cuadro 6.1: *Descripción de los parámetros usados en las tablas de validación*

Identificador	PA-1.
RF relacionado	RF-I-01
Descripción	La nueva interfaz se conecta al NameNode de Hadoop.
Precondiciones	1. Están configuradas las variables de entorno HADOOP_DEFAULT y HADOOP_PORT. 2. El NameNode de Hadoop se encuentra funcionando.
Procedimiento	1. La aplicación llama a la función MPI_Init_thread. 2. Se desconecta al llamar a la función MPI_Finalize.
Éxito	Si.

Cuadro 6.2: *Prueba de aceptación PA-1*

Identificador	PA-2.
RF relacionado	RF-I-01 RF-I-02 RF-I-03 RNF-I-01 RNF-I-03 RNF-I-04
Descripción	Se puede abrir y cerrar un fichero del sistema de ficheros HDFS a través de llamadas MPI.
Precondiciones	1. Están configuradas las variables de entorno HADOOP_DEFAULT y HADOOP_PORT. 2. El NameNode de Hadoop se encuentra funcionando. 3. En el sistema de ficheros de HDFS, se crea un fichero con el nombre de “prueba”.
Procedimiento	1. La función llama a MPI_Init_thread. 2. La aplicación llama a la función de MPI_File_open para abrir el fichero, indicando la ruta del fichero y comenzando esta por “hdfs:”. La apertura del fichero se hace en modo de lectura. 3. La aplicación llama a la función de MPI MPI_File_close para cerrar el fichero. 4. La aplicación llama a la función MPI_Finalize y, acaba su ejecución.
Postcondiciones	1. El proceso abre y cierra el archivo sin que se devuelva ningún error.
Éxito	Si.

Cuadro 6.3: *Prueba de aceptación PA-2*

Identificador	PA-3.
RF relacionado	RF-I-01 RF-I-02 RF -I-03 RF-I-04 RF-I-05 RF-I-06 RNF-I-01 RNF-I-03 RNF-I-04
Descripción	Funciones de escritura.
Precondiciones	1. Están configuradas las variables de entorno HADOOP_DEFAULT y HADOOP_PORT. 2. El NameNode de Hadoop se encuentra funcionando. 3. En el sistema de ficheros de HDFS se crea un fichero con el nombre de "prueba".
Procedimiento	1. La función llama a MPI_Init_thread. 2. La aplicación llama a la función de MPI_File_open para abrir el fichero, indicando la ruta del fichero y comenzando esta por "hdfs:". La apertura del fichero se hace en el modo de lectura. 3. La aplicación llama a la función MPI_File_read_at, a la que se le indica un offset de lectura y un número de bytes a leer. 4. La aplicación llama a la función MPI_Get_Count y se almacena el valor. 5. La aplicación llama a la función MPI_File_read, a la cual se le indica el número de bytes a leer. 6. La aplicación llama a la función MPI_Get_Count y se almacena el valor. 7. La aplicación llama a la función MPI_File_read_shared, a la cual el número de bytes a leer. 8. La aplicación llama a la función MPI_Get_Count y se almacena el valor. 9. La aplicación llama a la función de MPI MPI_File_close para cerrar el fichero. 10. La aplicación llama a la función MPI_Finalize y acaba su ejecución.
Postcondiciones	1. Se han leído del fichero el número de bytes indicados en cada una de las funciones indicadas.
Éxito	Si.

Cuadro 6.4: *Prueba de aceptación PA-3*

Identificador	PA-4.
RF relacionado	RF-I-01 RF-I-02 RF-I-03 RF-I-07 RNF-I-01 RNF-I-04
Descripción	Definir una vista sobre un fichero.
Precondiciones	1. Están configuradas las variables de entorno HADOOP_DEFAULT y HADOOP_PORT. 2. El NameNode de Hadoop se encuentra funcionando. 3. En el sistema de ficheros de HDFS, se crea un fichero con el nombre de "prueba".
Procedimiento	1. La función llama a MPI_Init_thread. 2. La aplicación llama a la función de MPI_File_open para abrir el fichero, indicando la ruta del fichero y comenzando esta por "hdfs:". La apertura del fichero se hace en el modo de lectura. 3. Se crea la vista para el fichero. 4. Se establece la nueva vista para dicho fichero por medio de la función MPI_File_set_view. 5. La aplicación llama a la función de MPI MPI_File_close para cerrar el fichero. 6. La aplicación llama a la función MPI_Finalize y acaba su ejecución.
Post condiciones	1. Se ha establecido la vista para el fichero sin que se produzca ningún error.
Éxito	Si.

Cuadro 6.5: *Prueba de aceptación PA-4*

Identificador	PA-5.
RF relacionado	RF-I-01 RF-I-02 RF-I-03 RF-I-08 RNF-I-01 RNF-I-03 RNF-I-04
Descripción	Prueba escritura en un fichero.
Precondiciones	1. Están configuradas las variables de entorno HADOOP_DEFAULT y HADOOP_PORT. 2. El NameNode de Hadoop se encuentra funcionando. 3. En el sistema de ficheros de HDFS, se crea un fichero con el nombre de "prueba".
Procedimiento	1. La función llama a MPI_Init_thread. 2. La aplicación llama a la función de MPI_File_open para abrir el fichero, indicando la ruta del fichero y comenzando esta por "hdfs:". La apertura del fichero se hace en el modo de escritura. 3. Se crea un buffer de escritura. 4. La aplicación llama a la aplicación MPI_File_write, a la cual se le indica un número de bytes a escribir y el buffer que se desea volcar en disco. 5. La aplicación llama a la función MPI_Get_Count y se almacena el valor. 6. La aplicación llama a la función de MPI MPI_File_close para cerrar el fichero. 7. La aplicación llama a la función MPI_Finalize y acaba su ejecución.
Post condiciones	1. Se han escrito en el fichero el número de bytes indicados. 2. El fichero se ha creado de nuevo y tiene el mismo tamaño que el buffer número de bytes escritos.
Éxito	Si.

Cuadro 6.6: *Prueba de aceptación PA-5*

Identificador	PA-6.
Requisito relacionado	RF-I-01 RF-I-09 RNF-I-01 RNF-I-03
Descripción	Eliminado de un fichero.
Precondiciones	1. Están configuradas las variables de entorno HADOOP_DEFAULT y HADOOP_PORT. 2. El NameNode de Hadoop se encuentra funcionando. 3. En el sistema de ficheros de HDFS, se crea un fichero con el nombre de "prueba".
Procedimiento	1. La función llama a MPI_Init_thread. 2. La aplicación llama a la función de MPI_File_delete indicando la ruta del fichero, empezando esta por "hdfs:". 3. La aplicación llama a la función MPI_Finalize y acaba su ejecución.
Post condiciones	1. El fichero ha sido eliminado.
Éxito	Si.

Cuadro 6.7: *Prueba de aceptación PA-6*

Identificador	PA-7.
Requisito relacionado	RF-I-01 RF-I-02 RF-I-03 RF-I-10 RNF-I-01 RNF-I-03 RNF-I-04
Descripción	Desplazar puntero interno de lectura.
Precondiciones	1. Están configuradas las variables de entorno HADOOP_DEFAULT y HADOOP_PORT. 2. El NameNode de Hadoop se encuentra funcionando. 3. En el sistema de ficheros de HDFS, se crea un fichero con el nombre de "prueba".
Procedimiento	1. La función llama a MPI_Init_thread. 2. La aplicación llama a la función de MPI_File_open e indicando la ruta del fichero, empezando esta por "hdfs:". La apertura del fichero se hace en el modo de escritura. 3. La aplicación llama a la función MPI_File_seek con el objetivo de desplazar el puntero un número de bytes. 4. La aplicación llama a la función MPI MPI_File_close para cerrar el fichero. 5. La aplicación llama a la función MPI_Finalize y acaba su ejecución.
Post condiciones	1. La función de desplazar el puntero ha devuelto el número de bytes desplazados .
Éxito	Si.

Cuadro 6.8: *Prueba de aceptación PA-7*

Identificador	PA-8.
RF relacionado	RF-I-01 RF-I-02 RF-I-03 RF-I-11 RNF-I-01 RNF-I-03 RNF-I-04
Descripción	Consulta del tamaño de un fichero.
Precondiciones	1. Están configuradas las variables de entorno HADOOP_DEFAULT y HADOOP_PORT. 2. El NameNode de Hadoop se encuentra funcionando. 3. En el sistema de ficheros de HDFS, se crea un fichero con el nombre de "prueba".
Procedimiento	1. La función llama a MPI_Init_thread. 2. La aplicación llama a la función de MPI_File_open indicando la ruta del fichero, empezando esta por "hdfs:". La apertura del fichero se hace en el modo de escritura. 3. La aplicación llama a la función MPI_File_get_size para conocer el tamaño del fichero. 4. La aplicación llama a la función MPI MPI_File_close para cerrar el fichero. 5. La aplicación llama a la función MPI_Finalize y acaba su ejecución.
Post condiciones	1. La función devuelve correctamente el tamaño del fichero.
Éxito	Si.

Cuadro 6.9: *Prueba de aceptación PA-8*

Identificador	PA-9.
Requisito relacionado	RF-I-01 RF-I-02 RF-I-03 RF-I-05 RF-I-11 RF-I-12 RNF-I-01 RNF-I-03 RNF-I-04
Descripción	Mimir Localidad.
Precondiciones	1. Están configuradas las variables de entorno HADOOP_DEFAULT y HADOOP_PORT. 2. El NameNode de Hadoop, se encuentra funcionando. 3. En el sistema de ficheros de HDFS se crea un fichero con el nombre de "prueba".
Procedimiento	1. Se modifica la librería de Mimir con el objetivo de mostrar cómo realiza el fraccionamiento de los ficheros. 2. Se ejecuta una prueba de Map-Reduce sobre el fichero "prueba", usando la librería de Mimir.
Post condiciones	1. La partición de los bloques se tiene que haber hecho por máquinas.
Éxito	Si.

Cuadro 6.10: *Prueba de aceptación PA-9*

Identificador	PA-10.
Requisito relacionado	RF-M-13 RNF-M-05 RNF-M-06
Descripción	Modificación de la función de Map y Reduce.
Precondiciones	1. Se está usando un entorno Linux. 2. Se dispone de un compilar de C, C++.
Procedimiento	1. Se modifican las funciones de Map y Reduce de la librería Mimir. 2. Se vuelve a compilar la aplicación completamente. 3. Se ejecuta la aplicación de nuevo usando el fichero "prueba" para comprobar su correcto funcionamiento.
Post condiciones	1. La modificación de las funciones de Map y de Reduce no influyen en el funcionamiento del sistema.
Éxito	Si.

Cuadro 6.11: *Prueba de aceptación PA-10*

Identificador	PA-11.
Requisito relacionado	RNF-M-05 RNF-M-06
Descripción	El sistema está implementado en C o C++.
Precondiciones	1. Se dispone de un compilador de C, C++. 2. Se usa un entorno Linux.
Procedimiento	1. Se compila la nueva versión de la interfaz. 2. Se compila la nueva versión de Mimir.
Post condiciones	1. La interfaz se compila correctamente. 2. La librería de Mimir se compila correctamente.
Éxito	Si.

Cuadro 6.12: *Prueba de aceptación PA-11*

Identificador	PA-12.
Requisito relacionado	RF-I-01 RF-I-02 RF-I-03 RF-I-04 RF-I-05 RF-I-06 RF-I-07 RF-I-08 RF-I-09 RF-I-10 RF-I-11 RF-M-12 RF-M-13 RNF-I-01 RNF-I-02 RNF-I-03 RNF-I-04 RNF-I-05 RNF-I-05
Descripción	Manejo de errores.
Precondiciones	1. Están configuradas las variables de entorno HADOOP_DEFAULT y HADOOP_PORT. 2. El NameNode de Hadoop se encuentra funcionando. 3. En el sistema de ficheros de HDFS, se crea un fichero con el nombre de "prueba".
Procedimiento	1. Se diseña una aplicación que llama a todas y cada una de las funciones implementadas. 2. Se asegura que las llamadas a dichas funciones, producen un fallo. 3. Se comprueba que el fallo devuelto por cada una de las funciones, sigue el estándar de MPI.
Post condiciones	1. Cada una de las funciones devuelve el código de error correcto.
Éxito	Si.

Cuadro 6.13: *Prueba de aceptación PA-12*

Para finalizar y comprobar que cada uno de los requisitos definidos se cumplen, se ha realizado la siguiente matriz (cuadro 6.14), en ella se asocia cada una de las diferentes pruebas con cada uno de los requisitos puntualizados para este proyecto.

-	PA-1	PA-2	PA-3	PA-4	PA-5	PA-6	PA-7	PA-8	PA-9	PA-10	PA-11	PA-12
RF-I-1	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓
RF-I-2		✓	✓	✓	✓		✓	✓	✓			✓
RF-I-3		✓	✓	✓	✓		✓	✓	✓			✓
RF-I-4			✓									✓
RF-I-5			✓						✓			✓
RF-I-6			✓									✓
RF-I-7				✓								✓
RF-I-8					✓							✓
RF-I-9						✓						✓
RF-I-10							✓					✓
RF-I-11								✓	✓			✓
RF-M-12									✓			✓
RF-M-13										✓		✓
RNF-I-01		✓	✓	✓	✓		✓	✓	✓			✓
RNF-I-02												✓
RNF-I-03		✓	✓		✓	✓	✓	✓	✓			✓
RNF-I-04		✓	✓	✓	✓		✓	✓	✓			✓
RNF-I-05										✓	✓	
RNF-M-06										✓	✓	

Cuadro 6.14: *Matriz pruebas aceptación*

6.1.2 Pruebas de verificación

Durante esta sección, se realizarán las pruebas de verificación para comprobar que el sistema desarrollado funciona correctamente.
Para estas pruebas se ha utilizado la siguiente tabla.

Identificador	Sirve para identificar de forma unívoca la prueba.
RF relacionado	Cada uno de los requisitos que se han verificado con esta prueba.
Descripción	Describe el objetivo de la prueba.
Precondiciones	Indican las condiciones iniciales del sistema.
Procedimiento	Indican los pasos que deben seguirse para replicar la prueba.
Postcondiciones	Indican las condiciones finales del sistema tras la prueba.
Éxito	Indica si el sistema a pasado la prueba satisfactoriamente.

Cuadro 6.15: *Descripción de los parámetros utilizados en las pruebas de verificación*

Identificador	PV-01
RF relacionado	RF-I-01 RF-I-02 RF-I-03 RF-I-04
Descripción	Lectura con offset explícito.
Precondiciones	1. Están configuradas las variables de entorno HADOOP_DEFAULT y HADOOP_PORT. 2. El NameNode de Hadoop se encuentra funcionando. 3. En el sistema de ficheros de HDFS, se crea un fichero con el nombre de "prueba", el cual contiene datos en formato de texto y un tamaño mínimo de 2MB.
Procedimiento	1.Se crea un buffer de 1MB, donde se van a almacenar los datos leídos. 2. Se llama a la función MPI_Init_thread para iniciar una aplicación de MPI. 3. Se llama a la función de MPI_File_open para abrir el fichero "prueba" en modo de lectura. 5. Se llama a la función de MPI_File_read_at la cual leerá 1 MB, saltándose los primeros 512KB del fichero. 6. Se llama a la función de MPI_Close para cerrar el fichero abierto previamente. 7. Se llama a la función MPI_Finalize para terminar la aplicación de MPI.
Postcondiciones	1. La aplicación ha leído la secuencia indicada.
Éxito	Si

Cuadro 6.16: *Prueba de verificación PV-01*

Identificador	PV-02
RF relacionado	RF-I-01 RF-I-02 RF-I-03 RF-I-05
Descripción	Lectura con puntero individual.
Precondiciones	1. Están configuradas las variables de entorno HADOOP_DEFAULT y HADOOP_PORT. 2. El NameNode de Hadoop se encuentra funcionando. 3. En el sistema de ficheros de HDFS, se crea un fichero con el nombre de "prueba", el cual contiene datos en formato de texto y de tamaño mínimo de 2 MB.
Procedimiento	1.Se crea un buffer de 1MB donde se van a almacenar los datos leídos. 2. Se llama a la función MPI_Init_thread para iniciar una aplicación de MPI. 3. Se llama a la función de MPI_File_open para abrir el fichero "prueba" en modo de lectura. 5. Se llama a la función de MPI_File_read que leerá los 512 KB iniciales del fichero. 6. Se llama a la función de MPI_File_read que leerá los siguientes 512 KB del fichero. 7. Se llama a la función de MPI_Close para cerrar el fichero abierto previamente. 8. Se llama a la función MPI_Finalize, para terminar la aplicación de MPI.
Postcondiciones	1. La aplicación ha leído la secuencia indicada.
Éxito	Si.

Cuadro 6.17: *Prueba de verificación PV-02*

Identificador	PV-03
RF relacionado	RF-I-01 RF-I-02 RF-I-03 RF-I-06
Descripción	Lectura con puntero compartido.
Precondiciones	1. Están configuradas las variables de entorno HADOOP_DEFAULT y HADOOP_PORT. 2. El NameNode de Hadoop se encuentra funcionando. 3. En el sistema de ficheros de HDFS se crea un fichero con el nombre de "prueba", el cual contiene datos en formato de texto y un tamaño de al menos 2 MB. 4. Dos procesos deben ejecutar esta aplicación.
Procedimiento	1. Se crea un buffer de 1MB donde se van a almacenar los datos leídos. 2. Se llama a la función MPI_Init_thread para iniciar una aplicación de MPI. 3. Se llama a la función de MPI_File_open para abrir un fichero no existente en modo de lectura. 4. Se llama a la función de MPI_File_read_shared, la cual leerá 512 KB del fichero. 5. Se llama a la función de MPI_File_read_shared, la cual leerá 512 KB del fichero. 6. Se llama a la función de MPI_Close para cerrar el fichero abierto previamente. 7. Se llama a la función MPI_Finalize, para terminar la aplicación de MPI.
Postcondiciones	1. La aplicación ha leído la secuencia indicada.
Éxito	Si.

Cuadro 6.18: *Prueba de verificación PV-03*

Identificador	PV-04
RF relacionado	RF-I-01 RF-I-02 RF-I-03 RF-I-07
Descripción	Lectura de un fichero con una vista definida.
Precondiciones	1. Están configuradas las variables de entorno HADOOP_DEFAULT y HADOOP_PORT. 2. El NameNode de Hadoop se encuentra funcionando. 3. En el sistema de ficheros de HDFS, se crea un fichero con el nombre de "prueba", el cual contiene datos en formato de texto y un tamaño mínimo de 2 MB.
Procedimiento	1. Se crea un buffer de 1MB donde se van a almacenar los datos leídos. 2. Se llama a la función MPI_Init_thread para iniciar una aplicación de MPI. 3. Se llama a la función de MPI_File_open para abrir un fichero no existente en modo de lectura. 4. Se define una vista sobre el fichero abierto de tal modo, que lea 512 KB, descarte los 512 KB siguientes y lea los siguientes 512 KB. 5. Se llama a la función de MPI_File_read la cual leerá 1 MB del fichero con la vista definida. 5. Se llama a la función de MPI_Close para cerrar el fichero abierto previamente. 6. Se llama a la función MPI_Finalize para terminar la aplicación de MPI.
Postcondiciones	1. La aplicación ha leído la secuencia indicada.
Éxito	Si.

Cuadro 6.19: *Prueba de verificación PV-04*

Identificador	PV-05
RF relacionado	RF-I-01 RF-I-02 RF-I-03 RF-I-08
Descripción	Escritura en un fichero.
Precondiciones	1. Están configuradas las variables de entorno HADOOP_DEFAULT y HADOOP_PORT. 2. El NameNode de Hadoop se encuentra funcionando.
Procedimiento	1. Se crea un buffer de 1MB, el cual se ha inicializado todo a '1'. 2. Se llama a la función MPI_Init_thread para iniciar una aplicación de MPI. 3. Se llama a la función de MPI_File_open para abrir un fichero no existente en modo de escritura. 4. Se llama a la función MPI_File_write para escribir en un fichero. Se le indica el buffer de escritura así como un tamaño de escritura de 512 KB. 5. Se llama a la función de MPI_Close para cerrar el fichero abierto previamente. 6. Se llama a la función MPI_Finalize para terminar la aplicación de MPI
Postcondiciones	1. Se ha creado el nuevo fichero de tamaño 512 KB y, además se ha escrito en él.
Éxito	Si.

Cuadro 6.20: *Prueba de verificación PV-05*

Identificador	PV-06
RF relacionado	RF-I-01 RF-I-02 RF-I-03 RF-I-09
Descripción	Borrado de un fichero.
Precondiciones	1. Están configuradas las variables de entorno HADOOP_DEFAULT y HADOOP_PORT. 2. El NameNode de Hadoop se encuentra funcionando. 3. En el sistema de ficheros de HDFS, se crea un fichero con el nombre de "prueba", el cual contiene datos en formato de texto y un tamaño mínimo de 2 MB.
Procedimiento	1. Se llama a la función MPI_Init_thread para iniciar una aplicación de MPI. 2. Se llama a la función de MPI_File_delete a la que se indica la ruta del fichero "prueba". 3. Se llama a la función MPI_Finalize para terminar la aplicación de MPI
Postcondiciones	1. El fichero ha sido eliminado.
Éxito	Si.

Cuadro 6.21: *Prueba de verificación PV-06*

Identificador	PV-07
RF relacionado	RF-I-01 RF-I-02 RF-I-03 RF-I-10
Descripción	Desplazamiento del puntero interno de lectura.
Precondiciones	1. Están configuradas las variables de entorno HADOOP_DEFAULT y HADOOP_PORT. 2. El NameNode de Hadoop se encuentra funcionando. 3. En el sistema de ficheros de HDFS, se crea un fichero con el nombre de "prueba", el cual contiene datos en formato de texto y un tamaño de al menos 2 MB.
Procedimiento	1.Se crea un buffer de 1MB, donde se van a almacenar los datos leídos. 2. Se llama a la función MPI_Init_thread para iniciar una aplicación de MPI. 3. Se llama a la función de MPI_File_open para abrir el fichero "prueba" en modo de lectura. 5. Se llama a la función MPI_File_seek para desplazar el puntero 512 KB desde el inicio del fichero. 6. Se llama a la función de MPI_File_read la cual leerá los siguientes 512 KB del fichero. 7. Se llama a la función de MPI_Close para cerrar el fichero abierto previamente. 8. Se llama a la función MPI_Finalize para terminar la aplicación de MPI.
Postcondiciones	1. Se ha desplazado el puntero interno de lectura. Y por lo tanto, los primeros 512 KB del fichero no se han leído.
Éxito	Si.

Cuadro 6.22: *Prueba de verificación PV-07*

Identificador	PV-08
RF relacionado	RF-I-01 RF-I-02 RF-I-03 RF-I-11
Descripción	Consulta del tamaño de un fichero.
Precondiciones	1. Están configuradas las variables de entorno HADOOP_DEFAULT y HADOOP_PORT. 2. El NameNode de Hadoop se encuentra funcionando. 3. En el sistema de ficheros de HDFS, se crea un fichero con el nombre de "prueba", el cual contiene datos en formato de texto y un tamaño mínimo de 2 MB.
Procedimiento	1.Se crea un buffer de 1MB, donde se van a almacenar los datos leídos. 2. Se llama a la función MPI_Init_thread para iniciar una aplicación de MPI. 3. Se llama a la función de MPI_File_open, para abrir el fichero "prueba" en modo de lectura. 4. Se llama a la función MPI_File_get_size, para obtener el tamaño del fichero. 5. Se llama a la función de MPI_Close para cerrar el fichero abierto previamente. 6. Se llama a la función MPI_Finalize, para terminar la aplicación de MPI.
Postcondiciones	1. El tamaño del fichero devuelto por la aplicación, es igual al tamaño real del fichero.
Éxito	Si.

Cuadro 6.23: *Prueba de verificación PV-08*

Identificador	PV-09
RF relacionado	RF-I-01 RF-I-02 RF-I-03 RF-I-05 RF-I-11 RF-M-12
Descripción	Uso de la librería de Mimir con la localidad en los ficheros implementada.
Precondiciones	1. Disponer de un fichero de 100MB en Hadoop, el cual tenga una replicación de 1.
Procedimiento	1. Ejecutar sobre ese fichero la operación de Map-Reduce, en la que no se ha implementado la localidad. 2. Ejecutar sobre ese fichero la operación de Map-Reduce, en la que se ha implementado la localidad.
Postcondiciones	1. Las dos operaciones tienen la misma salida de la operación de Map-Reduce.
Éxito	Si

Cuadro 6.24: *Prueba de verificación PV-09*

Identificador	PV-10
RF relacionado	RF-I-01 RF-I-02 RF-I-03 RF-I-05 RF-M-13
Descripción	Modificación de la función de Map de la librería de Mimir.
Precondiciones	1. Disponer de un fichero de 100MB en Hadoop, el cual tenga una replicación de 1. 2. Disponer de un compilador para C y para C++.
Procedimiento	1. Modificar la función de Map para que realice otra operación. 2. Compilar de nuevo la aplicación. 3. Ejecutar la nueva aplicación calculando previamente la salida esperada.
Postcondiciones	1 La nueva operación tiene la salida correcta.
Éxito	Si.

Cuadro 6.25: *Prueba de verificación PV-10*

A continuación, se detallan que pruebas de verificación confirman los diferentes tipos de requisitos (cuadro 6.26).

-	PV-1	PV-2	PV-3	PV-4	PV-5	PV-6	PV-7	PV-8	PV-9	PV-10
RF-I-1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
RF-I-2	✓	✓	✓	✓	✓		✓	✓	✓	✓
RF-I-3	✓	✓	✓	✓	✓		✓	✓	✓	✓
RF-I-4	✓									
RF-I-5		✓								
RF-I-6			✓							
RF-I-7				✓						
RF-I-8					✓					
RF-I-9						✓				
RF-I-10							✓			
RF-I-11								✓	✓	✓
RF-M-12									✓	
RF-M-13										✓

Cuadro 6.26: *Matriz pruebas verificación*

6.2 Estudio del rendimiento

En esta sección, se analizará el rendimiento de la interfaz desarrollada para MPI, en comparación al rendimiento de la interfaz nativa de HDFS escrita en java. Para las pruebas realizadas, se ha utilizado un cluster de 8 máquinas. Cada una de estas máquinas disponen de un procesador Intel Xeon E5405, 8 GB de RAM y un almacenamiento en disco de 1TB.

Para la realización de las pruebas, se ha montado una instalación de Hadoop en el cluster co-

mentado anteriormente. Por otro lado, en esta instalación, el NameNode se encuentra en el ordenador-1. Además, todos los ordenadores disponen de un datanode (figura 6-1).
Los aspectos destacados de dicha instalación se detallan a continuación:

- Tamaño del bloque: el tamaño del bloque ha sido establecido en 128MB.
- Factor de replicación: el factor de replicación se ha establecido en 1, por lo que cada bloque de datos solo se encuentra en un Datanode.
- Caché de Hadoop: tanto la caché de lectura como la de escritura, se borran cuando los datos son entregados a la aplicación.

Todas las pruebas siguientes, a excepción de las pruebas de escritura, se han realizado utilizando los mismos ficheros: uno de 100MB, otro de 1GB y otro de 10GB.

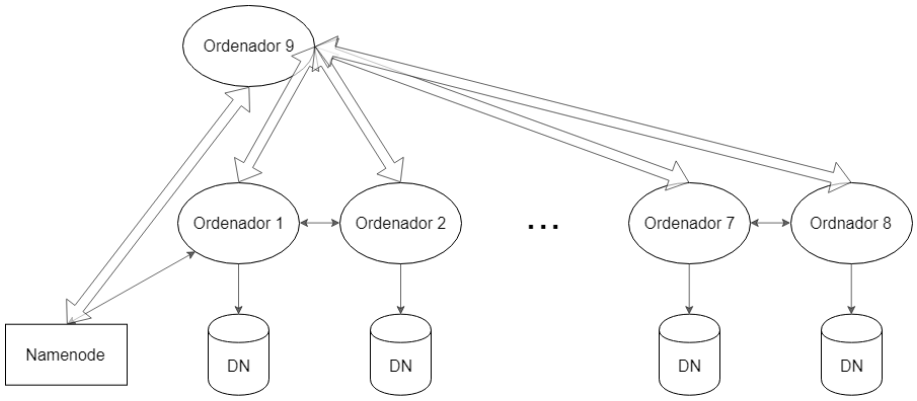

Figura 6-1: *Arquitectura del cluster para pruebas*

6.2.1 Comparativa entre la interfaz de Java e interfaz MPI

Con la realización de estas pruebas, se quiere comprobar la cantidad de sobrecarga que se produce entre el uso de la interfaz nativa de java y el uso de la interfaz desarrollada para MPI. Para medir el rendimiento de la nueva interfaz, se han definido tres tipos de pruebas. En primer lugar, se han realizado pruebas de lectura no paralelas (con un único proceso). En segundo lugar, se ha evaluado el rendimiento de la aplicación en lecturas paralelas, con 8 procesos leyendo del mismo fichero. En último lugar, se han realizado pruebas de escritura de un solo proceso en un único fichero. Respecto al tamaño del buffer de lectura y escritura, en estas pruebas, se han definido tres tamaños diferentes (128MB, 64MB y 1MB), de tal modo que se comprueba el rendimiento de la interfaz según el número de peticiones de acceso a disco que se realicen.

Lectura no paralela

En estas pruebas, se tiene un único proceso leyendo un fichero de forma secuencial. Para de evitar la localidad de los datos, se han ejecutado todas desde el ordenador-9 el cual no dispone de un datanode. Además, se han utilizado diferentes funciones de lectura: la función nativa de java para leer un fichero (Java), la función de lectura con un offset explícito (MPI-at), la función de lectura con puntero interno (MPI-read) y el uso de un puntero compartido (MPI-shared). Por último, se ha definido una vista sobre un fichero en la que el proceso accede a todo el fichero (MPI-view).
Los resultados obtenidos se muestran en las siguientes gráficas.

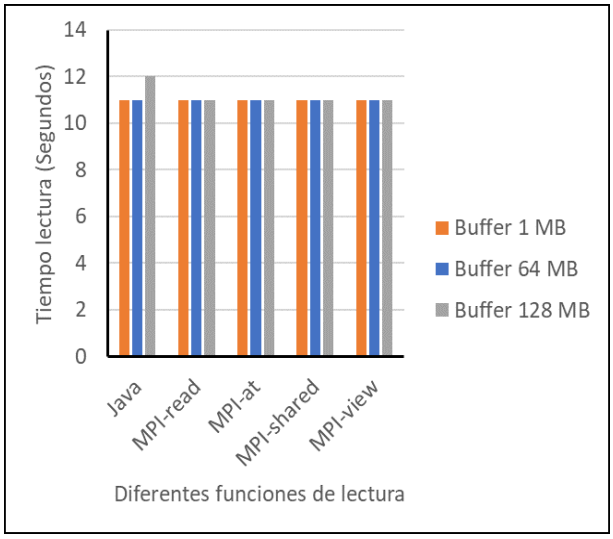

Figura 6-2: *Lectura 1 proceso fichero 100MB*

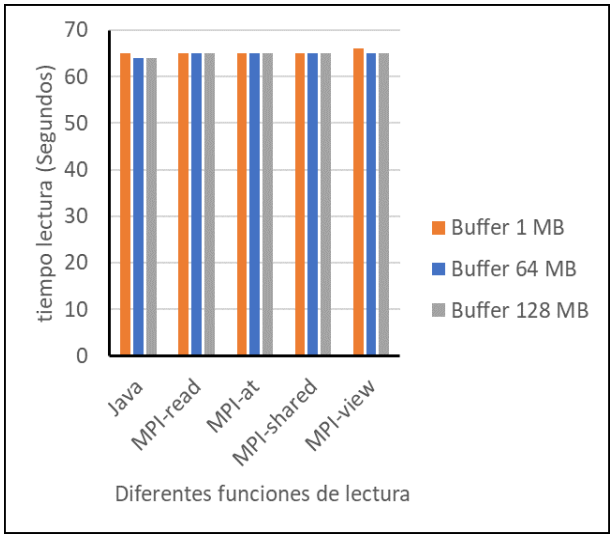

Figura 6-3: *Lectura un proceso fichero 1GB*

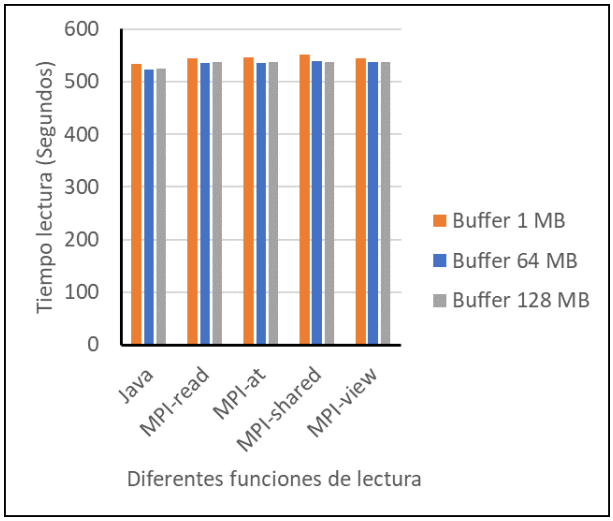

Figura 6-4: *Lectura un proceso fichero de 10GB*

Como se aprecia en las figuras anteriores (figuras 6-2, 6-3 y 6-4), el uso de la interfaz de MPI para acceder al sistema de ficheros de HDFS, incluye una pequeña sobrecarga en el tiempo de lectura, lo que se debe a que es más lento que el uso de java nativo, aunque se considera despreciable.
Respecto al tipo de función de MPI que se decida usar, se observa que en este caso es irrelevante, porque en todos los casos el tiempo de ejecución es similar. De igual modo, se observa que el número de peticiones de lectura apenas influye; aunque a más número de peticiones, peor es el rendimiento.

Lectura paralela

La siguiente prueba realizada, consiste en la lectura de un fichero de forma paralela. Para ello, se ha ejecutado en cada una de las 8 máquinas un proceso que lee un fragmento de un fichero de manera consecutiva. Al igual que en el caso anterior, con el objetivo de evitar la localidad de los datos, cada uno de los procesos java y MPI leen la misma porción del fichero, a excepción de la función de MPI_Read_shared, ya que por su funcionamiento, esto no se puede controlar.
Por otro lado, a excepción del puntero individual, se han utilizado los mismos tipos de lectura indicados en la sección anterior. Esto se debe a que los punteros de tipo individual no sirven para la lectura paralela de ficheros, a no ser, que se defina una vista sobre dicho fichero.

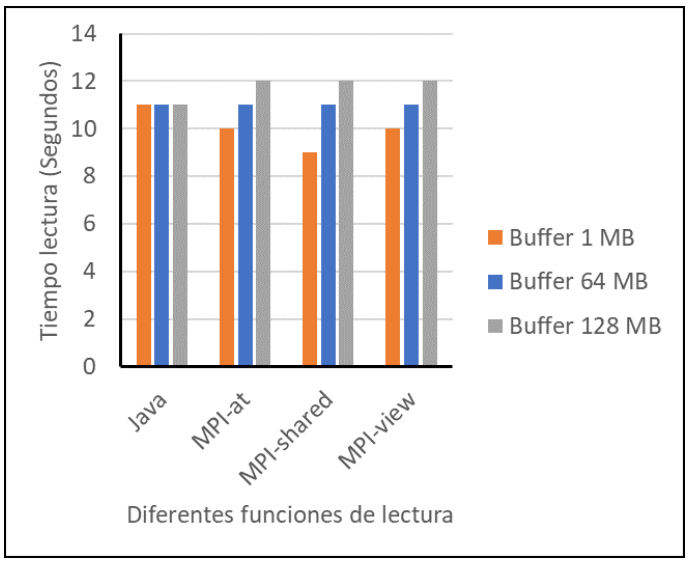

Figura 6-5: *Lectura 8 procesos fichero de 100MB*

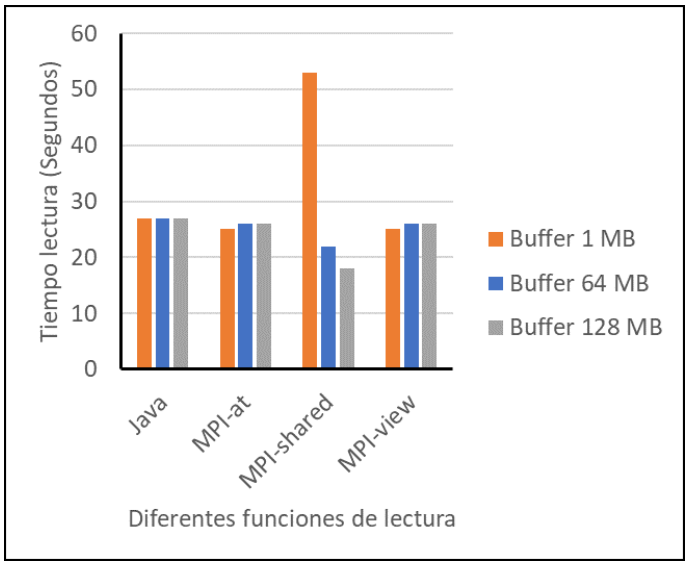

Figura 6-6: *Lectura 8 procesos fichero de 1GB*

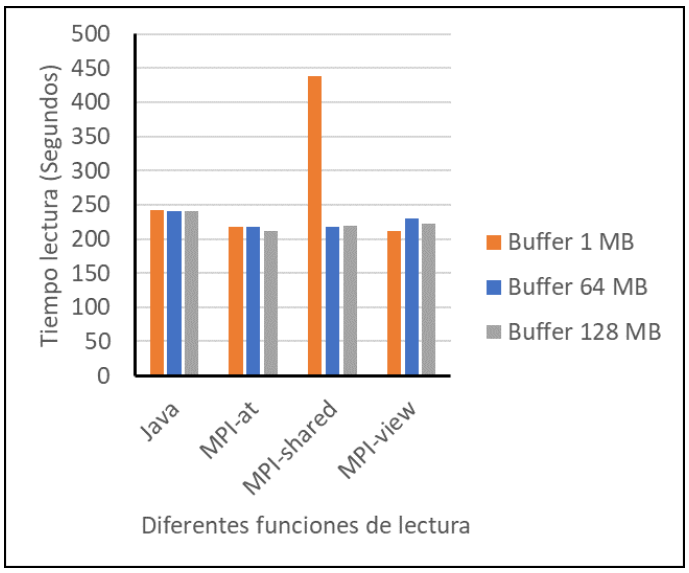

Figura 6-7: *Lectura de 8 procesos fichero de 10GB*

Como se observa en los gráficos (Figuras 6-5, 6-6 y 6-7) las lecturas paralelas usando MPI, a excepción del uso del puntero compartido, se comportan de forma similar que si se usa la interfaz de java. El tiempo global es menor cuando se usa MPI, como consecuencia del tiempo que emplea java en el uso de la memoria y en su recolector de basura.
Por otro lado como se observa en los gráficos (Figuras 6-5, 6-6 y 6-7), el puntero compartido, funciona bien cuando las peticiones de lectura son pocas pero, a mayor número de operaciones de lectura, peor es el comportamiento del mismo. Esto se debe a que cada vez que leen un fragmento de un proceso, deben comunicarse con el proceso 0 para que este les envíe el valor del puntero compartido.

Escritura no paralela

En esta sección, se muestran los resultados obtenidos en la escritura no paralela. La ejecución de estas pruebas, al igual que las pruebas de lectura, se realizan por un solo proceso ejecutado desde el ordenador-9. Como se indicó en la parte de análisis, solo se ha podido implementar la función de escritura MPI_File_write, por esto, solo se muestra esta función MPI en las figuras.

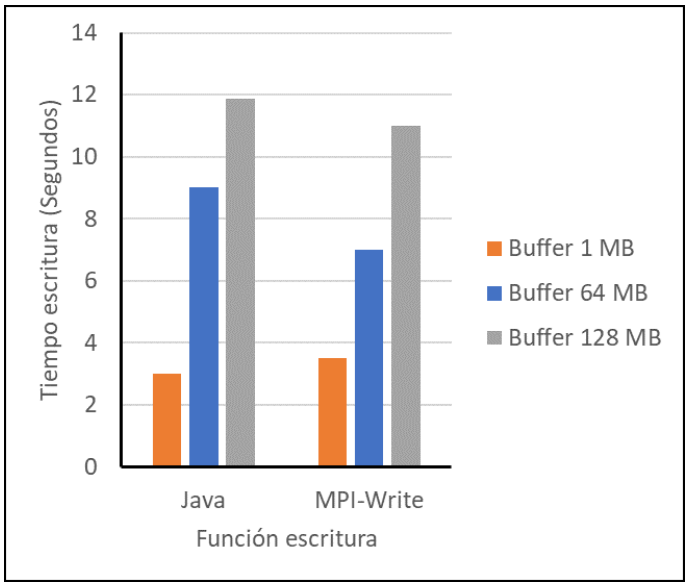

Figura 6-8: *Escritura 1 proceso fichero de 100MB*

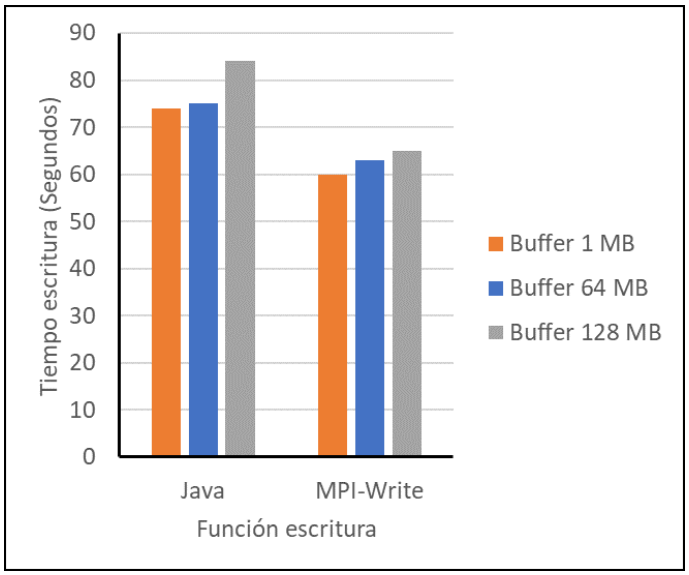

Figura 6-9: *Escritura 1 proceso fichero de 1GB*

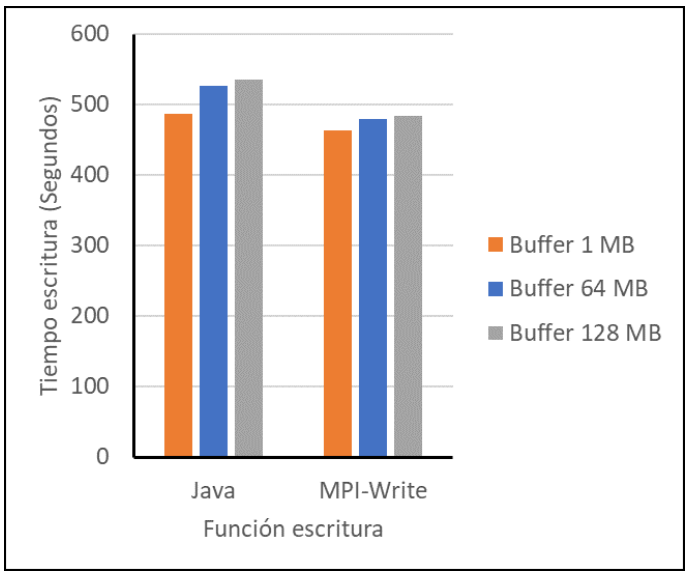

Figura 6-10: *Escritura 1 proceso fichero de 10GB*

Como se observa en las gráficas (Cuadros 6-8, 6-9 y 6-10), el tiempo de escritura usando la interfaz de MPI es menor. Esto es consecuencia de la gestión que hace java de la memoria y de su recolector de basura, puesto que como se observa en las figuras, a mayor sea el buffer de escritura en las pruebas, mayor es la diferencia entre los tiempos que tarda en escribir el fichero completo.
También se observa que, a menor tamaño de los bloques que se escriben en disco, mejor es el rendimiento de la aplicación.

6.2.2 Comparativa entre Mimir y Map-Reduce de Hadoop

Durante esta sección, se compara el rendimiento de la librería Mimir respecto al Map-Reduce de Hadoop. Para ello, se harán pruebas comparativas entre la ejecución de Map-Reduce de Hadoop y la ejecución de la librería de Mimir, tanto con la localidad implementada como sin ella. La ejecución de la librería se ha realizado de diferentes maneras: ejecutando un proceso en cada una de las ocho máquinas, ejecutando dos procesos en cada una de las máquinas y, ejecutando cuatro procesos en cada una de las ocho máquinas.
Además, se han ejecutado dos aplicaciones diferentes: una aplicación que cuenta el número de palabras disponibles en un fichero (WordCount), y otra que cuenta el número de veces que está una única palabra en un fichero (Grep).

WordCount

Esta aplicación cuenta el número de palabras que se encuentran en un fichero. Se han realizado pruebas con ficheros de 100MB, 1GB y 10GB; y con diferentes números de procesos (8, 16 y 32).

En cada una de las gráficas siguientes se compara el rendimiento de la aplicación, ejecutando con la localidad implementada y sin ella. Además, se incluye el tiempo que tarda Hadoop en realizar el mismo proceso.
En primer lugar, se incluye la gráfica comparativa realizando la operación de WordCount para un fichero de 100MB. En este caso, el paralelismo no es posible, ya que al haber un único bloque solo trabajan los procesos de una de las máquinas.

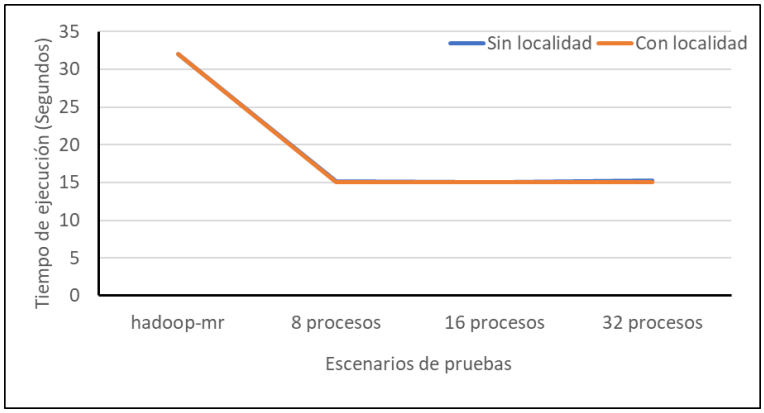

Figura 6-11: *Wordcount fichero de 100MB*

En este caso, el tiempo total de la aplicación de Mimir es un 50 % menor que el tiempo de Hadoop, debido a que Hadoop, dedica mucho tiempo a montar todo el proceso de Map-Reduce, es por esto por lo que no es aconsejable para analizar ficheros pequeños. En el caso de Mimir, como se muestra, en este caso la localidad no afecta en el resultado final.
En las siguientes gráficas se muestra la comparación entre el tiempo de lectura medio por cada proceso y el tiempo total de la aplicación.

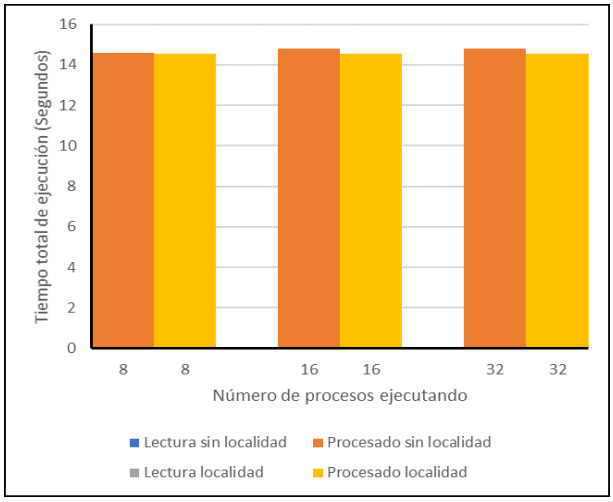

Figura 6-12: *Comparativa entre el tiempo de lectura y el tiempo de ejecución por cada proceso*

Como se aprecia en esta última gráfica (figura 6-12), el tiempo que usa la aplicación en leer es mínimo en comparación con el tiempo que usa para procesar los datos, es por ello, por lo que el tiempo entre incluir la localidad y no incluirla es el mismo.

En el siguiente gráfico (figura 6-13) se aprecia el tiempo de cómputo de una aplicación de WordCount con un fichero de 1GB.

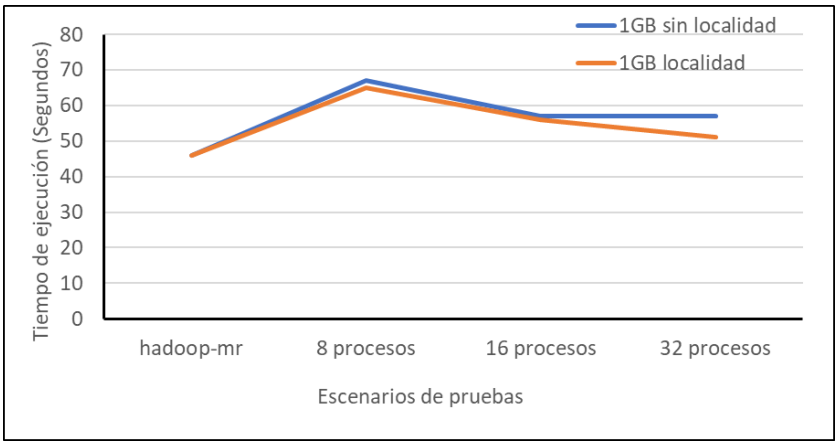

Figura 6-13: *WordCount fichero de 1GB*

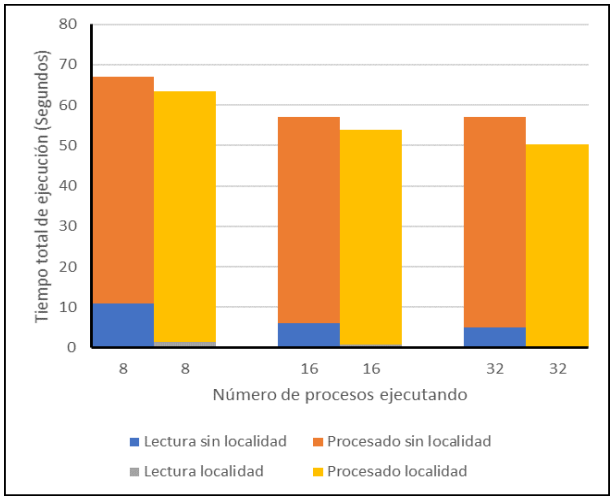

Figura 6-14: *Comparativa entre el tiempo de lectura y el tiempo de el tiempo ejecución por cada proceso*

En este caso, se puede observar cómo el tiempo medio de lectura por proceso se reduce en un 70 % al incluir la localidad en Mimir. A pesar de ello, el tiempo total de la aplicación no se reduce en la misma proporción, esto se debe, al igual que en el caso anterior, a que el tiempo de cómputo de la aplicación es mucho mayor que el tiempo de lectura.

En el siguiente gráfico (figura 6-15) se aprecia el tiempo de cómputo de una aplicación de WordCount con un fichero de 10GB.

En este caso, sí se observa la reducción del tiempo total de la aplicación gracias a la inclusión

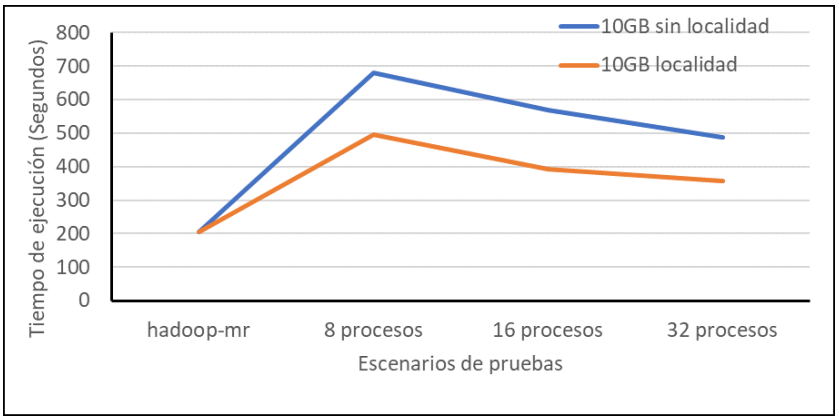

Figura 6-15: *WordCount fichero de 10GB*

de la localidad.

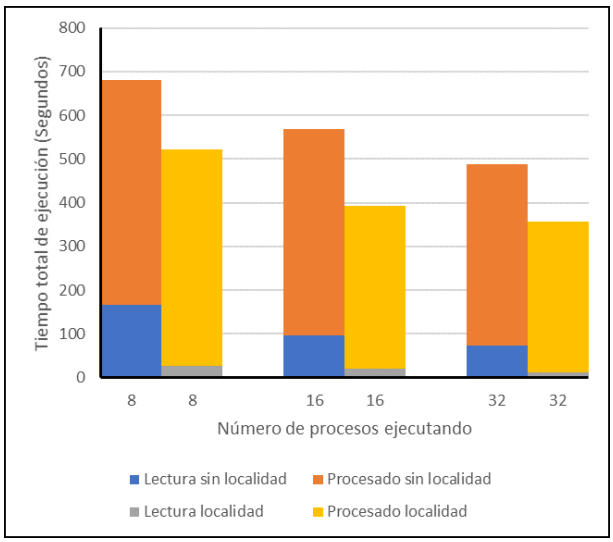

Figura 6-16: *Comparativa entre el tiempo de lectura y el tiempo de ejecución por cada proceso*

Como se observa en los resultados obtenidos, la localidad reduce el tiempo medio de lectura que tarda cada uno de los procesos en leer los datos. A pesar de ello, el tiempo en comparación con el Map-Reduce de Hadoop es bastante mayor, que en Mimir porque, la implementación interna de esta última pierde mucho tiempo en el procesado de los datos.

Como se indica en la ley de Amdahl: *la mejora obtenida en el rendimiento de un sistema debido a la alteración de uno de sus componentes está limitada por la fracción de tiempo que se utiliza ese componente* [34]. En esta implementación solo se mejoran lecturas, por ello, nuestra mejora se encuentra limitada únicamente por la fracción de tiempo de las lecturas.

Grep

Con el objetivo de reducir el tiempo de ejecución de la aplicación y comprobar que el tiempo de lectura mejora al implementar la localidad, se ha decido realizar un test con la aplicación Grep. Esta aplicación cuenta las veces que aparece una palabra en un texto. A pesar de que esta función lee el fichero completamente, al igual que en el caso anterior, el conjunto de datos generados por la función Map es inferior al conjunto generado por la función Map en la función de WordCount, y por lo tanto el tiempo de procesamiento es menor.

En el siguiente gráfico (figura 6-17) se aprecia el tiempo de cómputo de una aplicación de Grep con un fichero de 100MB.

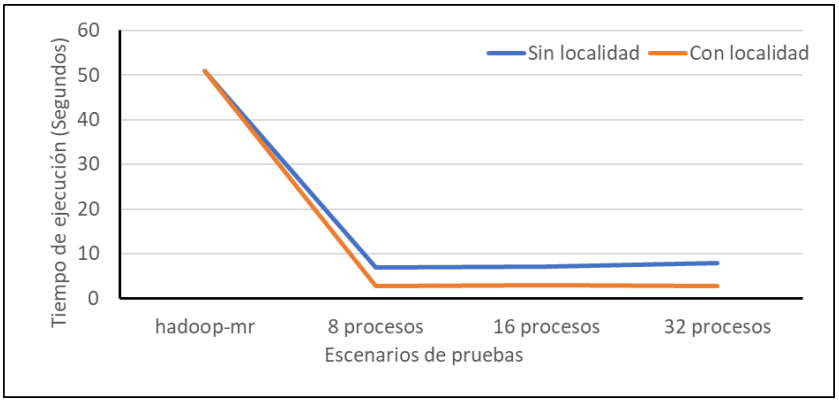

Figura 6-17: *Grep fichero de 100MB*

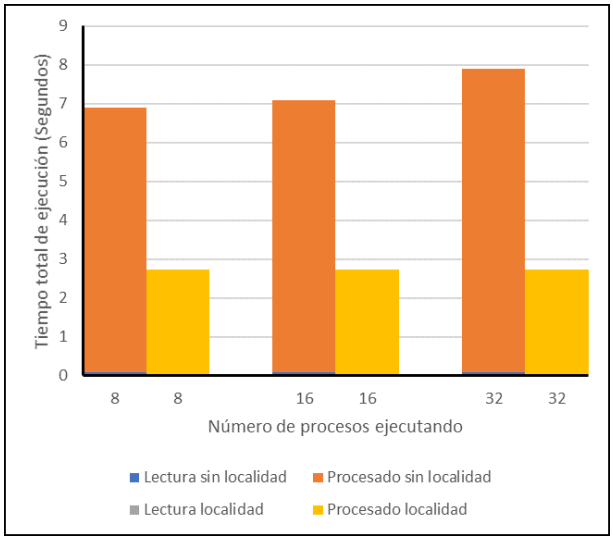

Figura 6-18: *Comparativa entre el tiempo de lectura y el tiempo de ejecución por cada proceso*

En esta prueba (figura 6-18), el tiempo de lectura es muy pequeño respecto al tiempo de ejecución de la aplicación, por este motivo, la mejora no es perceptible, incluso a mayor número

de procesos ejecutando el tiempo empeora.
En comparativa con Hadoop, el tiempo de ejecución se ha reducido en casi un 90 % en este caso. Al igual que en el caso de WordCount,esto se debe a que Hadoop necesita mucho tiempo en poner en funcionamiento el sistema. En el siguiente gráfico (figura 6-19) se aprecia el tiempo de cómputo de una aplicación de grep con un fichero de 1GB.

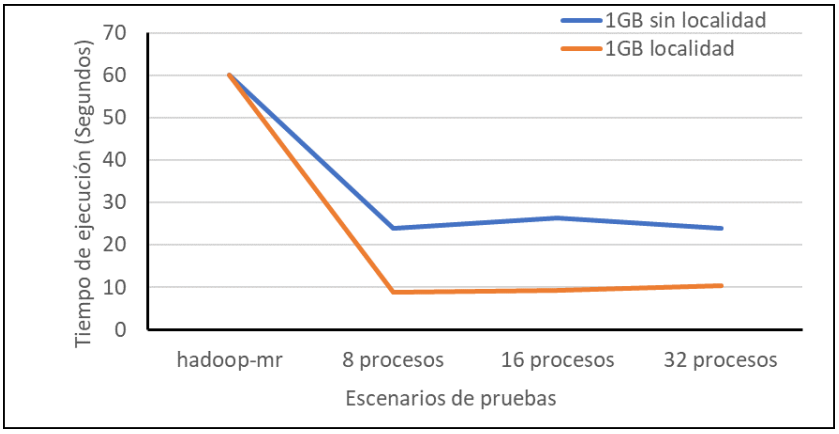

Figura 6-19: *Grep fichero de 1GB*

En este caso, el uso de la libería de Mimir con localidad presenta una clara mejora respecto a su uso sin localidad, concretamente una mejora de un 50 %.
En cambio Hadoop, tarda casi 600 % más, esto se debe, al igual que en el caso anterior, al tiempo que tarda en ponerse la aplicación de Map-Reduce en funcionamiento.

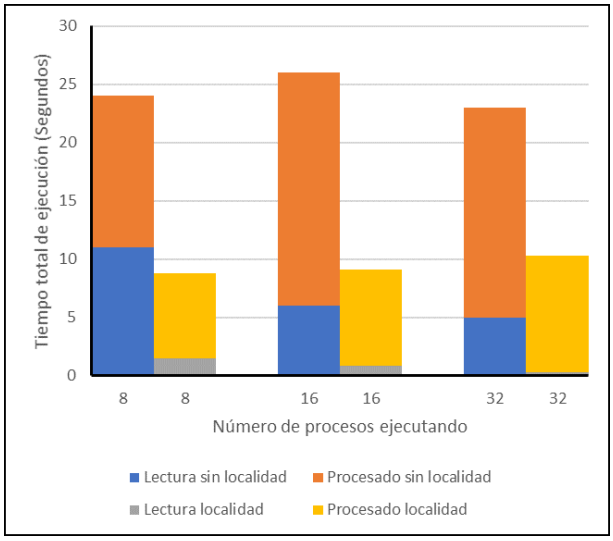

Figura 6-20: *Comparativa entre el tiempo de lectura y el tiempo de ejecución por cada proceso*

Como se observa, el tiempo medio de lectura por cada uno de los diferentes procesos, se ha reducido en un 70 % a la hora de implementar la localidad. En el siguiente gráfico (figura 6-19) se aprecia el tiempo de cómputo de una aplicación de Grep con un fichero de 1GB.

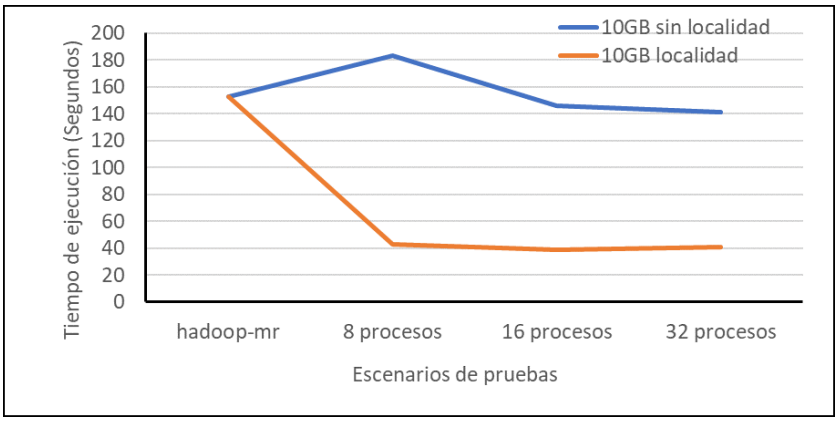

Figura 6-21: *Grep fichero de 10GB*

En este caso se muestra cómo Hadoop tarda un tiempo similar en procesar el fichero que la librería de Mimir sin incluir la localidad. Una vez que incluida la localidad, se comprueba cómo el tiempo de procesado disminuye notablemente, en concreto en un 70 %.

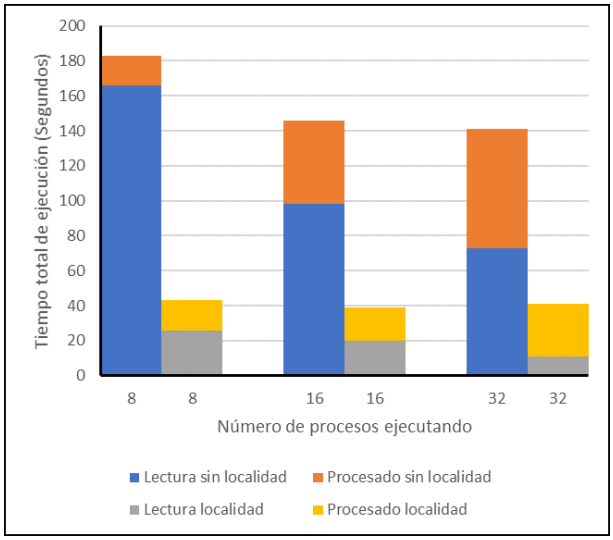

Figura 6-22: *Comparativa entre el tiempo de lectura y el tiempo ejecución por cada proceso*

En conclusión, en el WordCount, aunque el tiempo de lectura de mejora, no se mejora en gran medida el rendimiento de la función esto se debe a que la función de WordCount genera muchos datos de salida. Para mejorar el rendimiento de la librería, habría que mejorar Mimir desde el punto de vista del procesamiento, no desde el punto de vista de acceso a los datos como se ha realizado en este TFG. Sim embargo esto queda fuera del alcance de este proyecto.

Capítulo 7

Planificación y costes

En este apartado, se analiza en primer lugar la planificación que se ha llevado a cabo para la elaboración de este proyecto (sección 7.1), en segundo lugar se analizan los costes del mismo y se elabora un presupuesto (sección 7.2) y para finalizar en el tercer apartado se analiza el entorno socioeconómico (seccion 7.3).

7.1 Planificación

Durante esta sección se va a proceder a detallar la planificación que se ha llevado a cabo de este TFG. Debido a que este proyecto tiene 2 componentes principales, a la hora de realizar la planificación, los dos componentes se han tenido en cuenta. Los componentes de este proyecto son:

- Interfaz de MPI sobre HDFS
- Librería de Map-Reduce sobre MPI

Para estructurar, planificar y controlar el proceso de desarrollo de software es necesario el uso de una metodología de desarrollo. Para ello se han analizado tres modelos diferentes. El primero de ellos ha sido el modelo en cascada [35], en segundo lugar se ha analizado el modelo de prototipos[36] y por último el modelo en espiral [37]. El modelo en cascada consiste en un diseño secuencial, el cual consiste en una serie de etapas (análisis, diseño, implementación, pruebas y mantenimiento). Este modelo es sencillo, y se usa en proyectos de corta duración y poco innovadores, además, estos proyectos deben estar muy detallados, algo que no suele ser nada común, puesto que este proyecto, requiere tener, desde etapas tempranas del proyecto, el objetivo final del mismo no se ha podido utilizar para la planificación de este proyecto.
El segundo modelo que se analizó fue el modelo de prototipos, este modelo presenta el problema que necesita tener un prototipo inicial muy rápidamente, algo que en este proyecto, debido

a sus características no era posible ya que requería de una análisis inicial muy extenso.
Por estos motivos el mejor modelo a utilizar durante el desarrollo de este proyecto es el modelo en espiral, puesto que este fragmenta el proyecto en diferentes interacciones y se trata de un modelo iterativo. Este modelo presenta la desventaja de que es un modelo costoso. El modelo seleccionado, consta de una serie de etapas, las cuales se repiten una y otra vez, hasta que el proyecto ha sido desarrollado por completo. El modelo en espiral original definido por Boehm consta consta de 4 etapas, aunque en este proyecto se ha considerado más oportuno el uso de la variante de 6 etapas.

- Comunicación con el cliente: se obtienen los requisitos del sistema a desarrollar.
- Planificación: esta etapa determina los objetivos, alternativas y restricciones.
- Análisis: en esta etapa se analizan alternativas y se identifican los riesgos.
- Ingeniería: se completa el análisis y se realiza el diseño del sistema.
- Desarrollo y pruebas: se desarrolla el producto hasta el siguiente nivel.
- Evaluación del cliente: el cliente valora los resultados. El cliente evalúa el proyecto de una forma global.

Este proyecto consta de 5 iteraciones para su desarrollo. Las tres primeras,se corresponden con el desarrollo de la interfaz, mientras que las 2 últimas se corresponden con el desarrollo de la librería de Map-Reduce (Mimir).

- Primera iteración: funciones de manipulación de ficheros en el sistema de ficheros de HDFS.
- Segunda iteración: funciones de acceso a los ficheros
- Tercera iteración: funciones de definición de vistas en ficheros almacenados en HDFS.
- Cuarta iteración: desarrollo para que la librería de Mimir funcione con ficheros en HDFS
- Quinta iteración: implementación de la localidad en Mimir.

7.1.1 Tiempo estimado

El tiempo estimado para completar en este proyecto se estimó inicialmente en 8 meses. Este se empezó el día 15 de Octubre del 2017 y se finaliza el día 15 de Junio de 2018.
El siguiente diagrama de Gantt presenta el tiempo de vida del proyecto, en este diagrama se ha incluido una séptima etapa denominada documentación, la cual consiste en ir documentando cada uno de los pasos dados para la elaboración de este proyecto.

Nombre de la tarea	Inicio	Final	Duración
Iteración 1	**16-oct.-17**	**15-ene.-18**	**91**
Comunicación	16-oct.-17	20-oct.-17	4
Planificación.	20-oct.-17	30-oct.-17	10
Análisis.	30-oct.-17	4-dic.-17	35
Ingeniería.	4-dic.-17	18-dic.-17	14
Desarrollo y test	18-dic.-17	8-ene.-18	21
Evaluación del cliente	8-ene.-18	11-ene.-18	3
Documentación	11-ene.-18	15-ene.-18	4
Iteración 2	**15-ene.-18**	**27-feb.-18**	**43**
Comunicación	15-ene.-18	17-ene.-18	2
Planificación.	17-ene.-18	21-ene.-18	4
Análisis.	21-ene.-18	29-ene.-18	8
Ingeniería.	29-ene.-18	11-feb.-18	13
Desarrollo y test	11-feb.-18	19-feb.-18	8
Evaluación del cliente	19-feb.-18	22-feb.-18	3
Documentación	22-feb.-18	27-feb.-18	5
Iteración 3	**27-feb.-18**	**1-abr.-18**	**33**
Comunicación	27-feb.-18	1-mar.-18	2
Planificación.	1-mar.-18	5-mar.-18	4
Análisis.	5-mar.-18	10-mar.-18	5
Ingeniería.	10-mar.-18	17-mar.-18	7
Desarrollo y test	17-mar.-18	25-mar.-18	8
Evaluación del cliente	25-mar.-18	27-mar.-18	2
Documentación	27-mar.-18	1-abr.-18	5
Iteración 4	**1-abr.-18**	**14-may.-18**	**43**
Comunicación	1-abr.-18	3-abr.-18	2
Planificación.	3-abr.-18	6-abr.-18	3
Análisis.	6-abr.-18	20-abr.-18	14
Ingeniería.	20-abr.-18	28-abr.-18	8
Desarrollo y test	28-abr.-18	8-may.-18	10
Evaluación del cliente	8-may.-18	10-may.-18	2
Documentación	10-may.-18	14-may.-18	4
Iteración 5	**14-may.-18**	**18-jun.-18**	**35**
Comunicación	14-may.-18	17-may.-18	3
Planificación.	17-may.-18	19-may.-18	2
Análisis.	19-may.-18	21-may.-18	2
Ingeniería.	21-may.-18	24-may.-18	3
Desarrollo y test	24-may.-18	29-may.-18	5
Evaluación del cliente	29-may.-18	3-jun.-18	5
Documentación	3-jun.-18	18-jun.-18	15

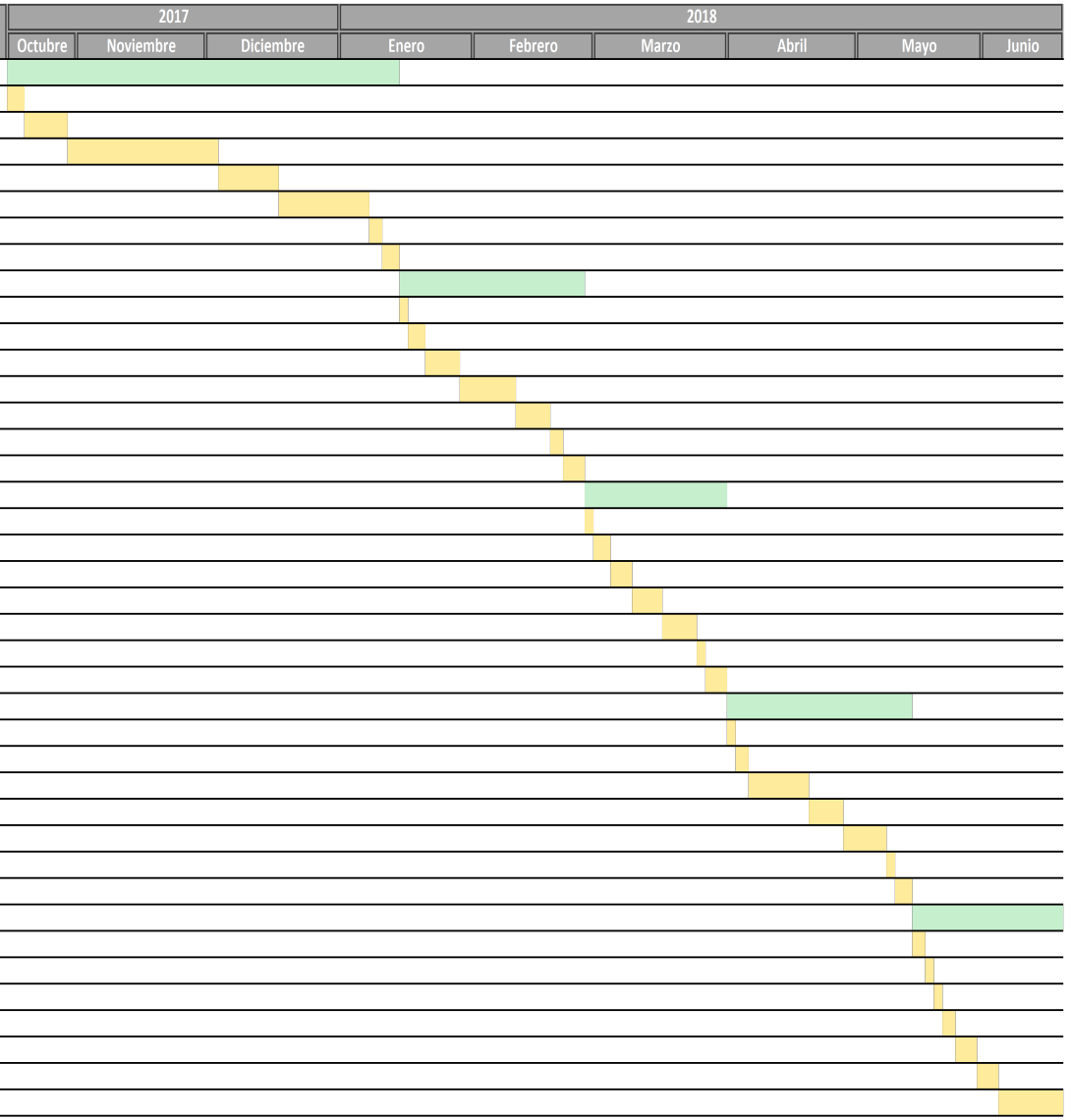

Figura 7-1: *Diagrama Gantt de planificación*

7.2 Presupuesto

En esta sección se detalla el presupuesto del proyecto. Por un lado se detallan los costes del proyecto y, por el otro se presenta la oferta presentada al cliente.

7.2.1 Costes del proyecto

La siguiente tabla muestra un resumen del presupuesto total del proyecto. Resumen del proyecto.

Autor	José Rivadeneira López-Bravo
Tutor	Félix García Carballeira
Duración	8 meses
Presupuesto	62.333,27 €

Cuadro 7.1: *Resumen proyecto*

Costes directos

En esta sección se presentan los costes directos del proyecto (cuadro 7.2). En primer lugar la siguiente tabla muestra los costes en personal del proyecto. El tutor de este TFG ha seguido el rol de director del proyecto, mientras que el estudiante ha seguido los roles de analista, desarrollador y el probador del proyecto.

Categoría	**Coste por hora**	**Horas**	**Total**
Director del proyecto	60	60	3.600 €
Analista	40	192	7.680 €
Desarrollador	25	312	7.800 €
Probador	25	125	3.125 €
Total			22.205€

Cuadro 7.2: *Costes en recursos humanos*

El cuadro 7.3 muestra los costes directos asociados a los equipos que han sido necesarios para la elaboración de este TFG. Para hallar dicho coste se ha establecido que los equipos informáticos tienen una vida media de un 5 años, por lo tanto se devalúan un 20 % anual. En base a esto y a su uso se ha estipulado la devaluación por uso que han sufrido en este proyecto.

Concepto	Coste inicial (€)	Dedicación, (meses)	Depreciación, (%)	Coste total,(€)
Ordenador sobremesa	1.100,00	8	14	154
Ordenador portátil	750,00	8	14	105
ARCOS Tucan	89.501,60	6	10	8.950.16
Impresora	100	8	14	14
Total				9.223,16

Cuadro 7.3: *Costes en equipamiento*

Cada uno de los equipos que se han utilizado se detallan en la siguiente lista:

- **Ordenador de sobremesa**: PC genérico I7 3770k, 16GB RAM, 1 TB.
- **Ordenador portátil**: MSI GL62 6QD, i7-6700HQ, 256 GB ssd, 1TB y 8GB RAM.
- **ARCOS TUCAN**: Clúster de 10 PC conectados en red, utilizado por el equipo de investigación Grupo de Aquitectura de Computadores, Universidad Carlos III de Madrid (ARCOS).
- **Impresora**: HP Desktop 1050.

En el siguiente cuadro 7.4 se muestran además otros costes directos, cómo es el material de oficina, se incluyen los bolígrafos, lapiceros, folios, tipex y subrayadores.

Concepto	Coste (€)
Material de oficina	96.25
Tinta(x2)	45

Cuadro 7.4: *Otros costes directos*

Resumen de costes asociados al proyecto

Una vez que se han calculado todos los costes del proyecto por separado se muestra un resumen de los mismos en el siguiente cuadro 7.5. Los costes indirectos se han calculado como el 20 % de los costes directos, entre ellos se incluye el consumo de agua, luz, teléfono y acceso a internet.

Concepto	Coste (€)
Recursos humanos	22.205
Equipos	9.223,16
Otros costes directos	96.25
Costes indirectos	6.634,9
Total	38.159,30

Cuadro 7.5: *Resumen de costes*

7.2.2 Oferta del proyecto

Durante esta sección se incluye una oferta completa del proyecto 7.6. En esta se incluye el riesgo estimado (15 %), el beneficio (20 %) y el IVA [38], correspondiente con el (21 %) actual en España.

Concepto	Coste
Riesgo (15 %)	5.723,90
Beneficio (20 %)	7.631,90
IVA (21 %)	10.818,17
Presupuesto total del proyecto	62.333,70

Cuadro 7.6: *Oferta completa del proyecto*

7.3 Entorno socio económico

En la actualidad, como se ha comentado en el presente documento, las empresas necesitan el procesado de grandes cantidades de datos en un tiempo finito para sus planes de negocio, por este motivo surgió la tecnología de Big Data. Esta permite el procesado de grandes conjuntos de datos, en un tiempo finito. Para ello usa grandes clusteres donde procesa los datos de una manera distribuida. Una de las principales aplicaciones que surgieron en este campo fue Hadoop y su aplicación de Map-Reduce, así como su sistema de ficheros HDFS.
Por otro lado, surgió la HPC, la cual surgió no con el objetivo de procesar datos, si no, con el objetivo de trabajar en entornos científicos resolviendo ecuaciones complejas en un tiempo finito, al igual que en el caso anterior, estos cálculos se realizan de una forma distribuida. Dentro de la HPC una de sus principales aplicaciones es MPI, la cual permite la comunicación y sincronización entre diferentes procesos.
Este proyecto se integra dentro de ambas tecnologías, ya que por un lado permite el uso de un sistema de ficheros como es HDFS dentro de aplicaciones de MPI y, por otro lado permite la ejecución de aplicaciones de Map-Reduce por medio de aplicaciones de HPC, en concreto usando aplicaciones MPI.

De este modo, se proporciona a las empresas una alternativa al uso del framework de Hadoop, para la creación y ejecución de aplicaciones Map-Reduce.

Capítulo 8

Conclusiones y trabajo futuro

Durante este capítulo se analizarán las conclusiones (sección 8.1) obtenidas durante la elaboración de este TFG. Así como las futuras líneas de trabajo que quedan abiertas tras su realización. (sección 8.2).

8.1 Conclusiones

Durante esta sección se hará una evaluación de los objetivos planteados en el incio del proyecto, así como un análisis posterior sobre si se han logrado con éxito o no.
El primer objetivo definido en este proyecto consiste en la elaboración de una interfaz para HDFS en MPI. Para la elaboración de esta interfaz, fue necesario, en primer lugar, realizar una investigación, para conseguir incluir una interfaz nueva para un sistema de ficheros en MPI,ya que no existe una buena documentación del proceso que se debe seguir.
Este objetivo se ha conseguido satisfactoriamente, como se ha presentado a lo largo de los capítulos de este documento. Hay que indicar que, la implementación realizada no soporta todas las funciones de MPI, como consecuencia de decidir usar la semántica de HDFS,es decir, solo se han implementado aquellas funciones que permite HDFS.

El segundo objetivo principal del proyecto, ha sido la modificación de una librería de Map-Reduce en MPI con el objetivo de que use la nueva interfaz desarrollada. Para este trabajo, debido a que la librería utilizada (Mimir) se encuentra aún en fase de desarrollo, ha sido necesario un análisis concienzudo sobre su funcionamiento, además de estudiar su posible optimización. Esta optimización de la librería ha sido posible por la incorporación del concepto de localidad, consiguiendo que cada uno de los datos que han sido guardados en HDFS, se procesen en las máquinas en las que se encuentran guardados y almacenados. Gracias a este logro, como se ha demostrado en los capítulos anteriores1, se consiguen reducir los tiempos medios de lectura

por cada uno de los procesos.

De cara a los usuarios del proyecto final, se ha desarrollado, una nueva forma de realizar aplicaciones de Map-Reduce sobre el sistema de ficheros de HDFS, sin tener que usar el framework de Hadoop.

8.2 Trabajo futuro

Tras la finalización de este TFG, se han abierto unos trabajos futuros, detallados en la siguiente lista:

- Implementación de un parche, para incluir fácilmente la implementación realizada de la interfaz dentro de MPICH con el objetivo de que aquellos usuarios que la deseen usar, puedan hacerlo.
- Optimización de la librería de Mimir. Como se ha visto, el procesamiento de los datos en Mimir aún es bastante lento, aunque, durante este trabajo se ha conseguido optimizar el acceso a disco de esta librería (siempre que se use el sistema de ficheros de HDFS).
- Evaluación del rendimiento en un entorno cloud, como podría ser Amazon EC2.

Glosario

ADIO Interfaz abstracta paralela, la cual provee un conjunto de operaciones de entrada y salida para sistemas de ficheros paralelos. 15, 16, 20, 33, 34

Apache Software Foundation Organización caritativa de los Estados Unidos sin ánimo de lucro, la cual desarrolla proyectos de software libre. 2, 9, 30, 120

Big Data Conjunto de técnicas distribuidas que permiten el tratamiento de grandes conjuntos de datos. 1, 2, 5, 6, 16, 96

C Lenguage de programación diseñado en los años 60. 30, 51, 66, 77

C++ Lenguage de programación basado en c del cual es su sucesor. 30, 66, 77, 120

Cluster Conjunto de ordenadores, unidos entre si por medio de una red de alta velocidad. 9

framework Entorno de trabajo estandarizado, donde se definen las prácticas y criterios para enfrentar y resolver problemas. 1, 2, 9, 11, 16, 30, 97, 100, 120, 125

Google Multinacional americana especializada en proveer servicios de internet, asi como son las búsquedas de información, o el procesado de los datos. 1, 2, 7, 9, 11, 120

Hadoop Framework que permite el procesamiento distribuido de grandes conjuntos de datos. vii, 2, 3, 9–11, 16, 18, 24, 30, 51, 53, 54, 57, 60–65, 67, 69–78, 84, 85, 87, 89, 90, 96, 97, 100, 107, 108, 111–113, 115, 116, 118–120, 123–125

Hadoop MapReduce Aplicaciones de MapReduce realizadas usan el framework de Hadoop. 9

IBM Spectrum Implementación de mpi, desarrollada por la empresa IBM. Esta implementación es comercial y por lo tanto no es de código libre. 21

IVA Impuesto sobre el valor añadido de España. 96

Java Lenguaje de programación de propósito general y orientado a objetos. 9, 11, 51, 59, 77, 78, 80, 82, 107, 120, 121

LibHdfs Librería para el lenguage de programación c que provee una api para manipular ficheros de HDFS sin usar el lenguage de programación de Java. 51

libjvm Librería de la máquina virtual de java. 51

Map Función dentro de Map-Reduce que permite procesar un par (clave, valor) para generar un conjunto de datos intermedio de forma (clave, valor). 8, 16–18, 23, 28, 57, 66, 77, 88, 102

Map-Reduce Es un modelo de programación que permite procesar y generar grandes conjuntos de datos, por medio principalmente de dos funciones (Map y Reduce). 1–3, 7, 8, 11, 16, 19, 20, 22, 23, 51, 57, 65, 76, 84, 85, 87, 89, 91, 92, 96, 97, 99, 100, 102, 110, 112, 119, 120, 123, 125

Mimir Biblioteca de map-reduce, la cual, funciona por medio de la biblioteca de paso de mensajes (MPI). xi, 3, 16–18, 22–24, 30, 33, 47, 49, 51, 55–57, 59, 65, 66, 76, 77, 84–87, 89, 90, 92, 99, 100, 120, 123–125

MPI-IO Porhacer. 2, 12–15, 19, 20, 29, 37, 53

MPICH Implementación de MPI, la cual permite correr aplicaciones de manera distribuida. 2, 12, 15, 16, 21, 22, 29, 30, 33, 100, 113, 114, 117

mr-mpi Biblioteca de map-reduce, la cual, funciona por medio de la biblioteca de paso de mensajes (MPI). 16, 22

NameNode Componente de Hadoop, en el cual se almacena toda la información del sistema de ficheros de HDFS, como el nombre de los ficheros, el árbol de directorios o el lugar donde se encuentran almacenados los bloques de los ficheros. 53, 60–65, 67, 69–76, 78, 109, 111, 118

Open-mpi Implementación de MPI de código libre, la cual es mantenida por un consorcio de universidades. 21

Reduce Función dentro de Map-Reduce la cual combina todos los valores intermedios generados por la función Map que tienen la misma clave intermedia, y devuelve el resultado final de la operación. 8, 16–18, 23, 28, 66, 102

Romio Implementación del accesso a los ficheros en el desarrollo realizado en MPICH. 15, 33, 113

software libre Software en el cual los usuario pueden ejecutarlo, distribuirlo, copiarlo, modificarlo e incluso mejorarlo, de una forma totalmente gratuita. 9, 30, 31

Unix Sistema operativo portable, multitarea y multiusuario. 9

URL Dirección de una página web. 8

Yahoo Empresa global con sede en Estados Unidos, la cual provee servicios en internet, como son búsquedas en internet, proveer correo. 2, 9, 11

Siglas

ARCOS Grupo de Aquitectura de Computadores, Universidad Carlos III de Madrid. 95

GFS Google File System. 1, 7, 9

HDFS Sistema de ficheros distribuido de Hadoop. vii, 2, 3, 9, 10, 19–21, 25, 27, 29, 30, 33–47, 51–56, 60–65, 67, 69–72, 74–77, 80, 91, 92, 96, 99, 100, 111–113, 115–118, 120, 125

HPC computación de altas prestaciones. 2, 3, 5, 12, 16, 96

JNI Interfaz nativa de java. 51

MPI Interfaz de paso de mensajes. 2, 3, 12, 13, 15, 16, 19–22, 24, 25, 34, 36, 37, 47, 52–54, 56, 59–65, 67, 69–78, 80, 82, 84, 91, 96, 99, 102, 113, 116, 117, 120, 121, 125

MPI-IO Interfaz de paso de mensajes, aplicada al acceso a los sistemas de ficheros. 16, 19, 23, 33, 34, 51, 116, 117

Posix Portable Operating System Interface. 19, 22, 47, 56, 120

TFG Trabajo de fin de grado. 3, 12, 19, 30, 90, 91, 94, 99, 100

Apéndice A

Instalación de Hadoop

En este anexo se va a realizar una explicación sobre como instalar Hadoop en un cluster. Se supondrá que se quiere instalar Hadoop en un total de tres máquinas. En la primera máquina se dispondrá del *namenode* y de un *datanode*; mientras que en la segunda y tercera máquina se dispondrá de un *datanode*. Esta documentación es válida para la versión 2.9.1 de Hadoop.

A.1 Antes de empezar

Antes de empezar con la instalación hay que verificar que se cumplen los siguientes requisitos.

1. En primer lugar hay que asegurarse que se tienen todas las máquinas montadas como un cluster, y que se pueden conectar entre ellas por medio del comando ssh [39].

2. Java debe estar configurado e instalado en todas las máquinas, así como, la variable de entorno *JAVA_HOME* debe estar establecida con la ruta en la cual se encuentra instalado java. En este caso usaremos la versión java-8, la cual se encuentra por defecto en la ruta /usr/lib/jvm/java-8-openjdk-amd64/jre.

A.2 Arquitectura del cluster de Hadoop

- Namenode: este componente es el encargado de gestionar el sistema de ficheros y de conocer donde se encuentra almacenado cada uno de los bloques de los ficheros.
- Datanode: este componente es el encargado de almacenar los bloques de un fichero.
- ResourceManager: administra las aplicaciones yarn que se están ejecutando.
- NodeManager: administra la ejecución de tareas en el nodo.

A.3 Descargar binarios de Hadoop

En primer lugar para comenzar con la instalación de Hadoop hay que descargar los binarios. Para ello, hay que seguir los siguientes comandos.

1. Usar el comando `cd` para dirigirnos al directorio donde queremos instalar Hadoop.
2. Usar el comando el siguiente comando para obtener los binarios de Hadoop.

   ```
   wget http://apache.rediris.es/hadoop/common/hadoop-2.9.1/hadoop-2.9.1.tar.gz
   ```

3. Usar el comando `tar -xzf hadoop-2.9.1.tar.gz` para descomprimir el archivo descargado.
4. Cambiar el nombre de la carpeta de hadoop-2.9.1 al nombre deseado con el comando `mv`, en este caso se cambiará el nombre por “hadoop". Durante siguientes secciones de este capítulo se supondrá que al poner una ruta que empiece por hadoop, se referirá a la ruta completa hasta esta carpeta.
5. Por último, establecer la variable de entorno PATH introduciendo las carpetas de instalación de Hadoop. hadoop/bin y hadoop/sbin.

Además de la variable de entorno PATH y JAVA_HOME explicadas anteriormente, se recomienda establecer las siguientes.

- HADOOP_INSTALL: Ruta completa a la carpeta de Hadoop descargada anteriormente.
- HADOOP_MAPRED_HOME: $HADOOP_INSTALL
- HADOOP_COMMON_HOME: $HADOOP_INSTALL
- HADOOP_HDFS_HOME: $HADOOP_INSTALL
- HADOOP_YARN_HOME: $HADOOP_INSTALL
- HADOOP_COMMON_LIB_NATIVE_DIR: $HADOOP_INSTALL/lib/native
- HADOOP_OPTS: “-Djava.library.path= $HADOOP_INSTALL/lib"

Con estas variables de entorno ya podremos ejecutar Hadoop sin problemas. En el caso de que se quieran ejecutar aplicaciones de C que hacen uso de la libreria “libhdfs"hay que configurar esta librería en la variable de entorno de LD_LIBRARY_PATH.
Para ello hay que escribir en el /.bahsrc
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/$HADOOP_INSTALL/lib/native.
Puesto que esta librería llama internamente a java hay que configurar también el classpath con los .jar correspondientes, para que funcione. Para incluir todos los ficheros .jar de Hadoop en el classpath seguimos los siguientes pasos:

1. En primer lugar ejecutar el siguiente comando: `hadoop classpath --glob >>.bashrc` . Con esto insertaremos las rutas a todos los .jar en el fichero .bashrc.

2. Abrimos el .bashrc y se añade justo antes de los .jar insertados anteriormente: CLASS_PATH=$CLASS_PATH:Rutas_a_los_jar... .

A.4 Configurar la instalación de Hadoop

Durante esta sección vamos a configurar cada una de las máquinas, eligiendo donde se va a instalar el NameNode, los datanode, el resource manager y los nodemanager.
En primer lugar hay que modificar el fichero hadoop/etc/hadoop/hadoop-env.sh. En el hay que incluir la siguiente linea al final: export JAVA_HOME= “Ruta de instalación de java", por defecto /usr/lib/jvm/java-8-openjdk-amd64/jre.
Ahora se va a proceder a señalar en que máquina se ejecuta el namenode, para ello se modifica el fichero: /hadoop/etc/hadoop/core-site.xml. En este fichero se define la propiedad fs.default.name, en la cual se indica la máquina donde se ejecuta el namenode y su puerto.

```
0  <?xml version="1.0" encoding="UTF-8"?>
1  <?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
2      <configuration>
3          <property>
4              <name>fs.default.name</name>
5              <value>hdfs://node-master:9000</value>
6          </property>
7      </configuration>
```

En el fichero hadoop/etc/hadoop/hdfs-site.conf, se indican tres nuevas propiedades.

- dfs.namenode.name.dir: Indica la ruta donde se almacenará la información del sistema de ficheros. Nótese que la ruta debe existir y se deben tener permisos de escritura en ella.
- dfs.datanode.data.dir: Indica la ruta donde se almacenarán los datos de los ficheros. Nótese que la ruta debe existir y se deben tener permisos de escritura en ella.
- dfs.replication: Indica la replicación por defecto que tendrán los nuevos ficheros que se creen dentro del sistema de ficheros. No se puede configurar una replicación mayor que el número de datanodes disponibles.

```
0  <configuration>
1      <property>
2              <name>dfs.namenode.name.dir</name>
```

```
3                 <value>/home/hadoop/data/nameNode</value>
4             </property>
5
6             <property>
7                 <name>dfs.datanode.data.dir</name>
8                 <value>/home/hadoop/data/dataNode</value>
9             </property>
10
11            <property>
12                <name>dfs.replication</name>
13                <value>1</value>
14            </property>
15        </configuration>
```

El siguiente fichero de configuración que modificaremos será el fichero: mapred-site.xml, este fichero piede venir guardado como mapred-site.xml.template, en tal caso hay que renombrarlo quitando el .template del final.
En este fichero se indicará que el framework por defecto que queremos utilizar para las operaciones de Map-Reduce es Yarn.

```
0         <configuration>
1             <property>
2                 <name>mapreduce.framework.name</name>
3                 <value>yarn</value>
4             </property>
5         </configuration>
```

El siguiente paso, consiste en la configuración del framework Yarn, para ello, hay que incluir las siguientes propiedades en el fichero yarn-site.xml.
Además sustituir el valor node-master por el valor indicado.

```
0         <configuration>
1             <property>
2                 <name>yarn.acl.enable</name>
3                 <value>0</value>
4             </property>
5
6             <property>
7                 <name>yarn.resourcemanager.hostname</name>
8                 <value>node-master</value>
9             </property>
```

```
10
11          <property>
12                  <name>yarn.nodemanager.aux-services</name>
13                  <value>mapreduce_shuffle</value>
14          </property>
15      </configuration>
```

Por último solo nos queda configurar los nodos esclavos, para ello hay que crear el fichero "slaves" dentro de la ruta hadoop/etc/hadoop/. En este fichero hay que poner el nombre de las máquinas que queremos que sean exclavas, es decir, aquellas en las que se encontrarán los datanode. Si queremos que el nodo principal (en el que se ejecuta el NameNode) disponga también de un datanode hay que incluir esta máquina dentro de las máquinas esclavas.
Tras esto, hay que copiar los binarios de Hadoop, así como todos los ficheros de configuración en cada una de las máquinas esclavas.

A.5 Ejecutando Hadoop

Con los pasos realizados anteriormente la configuración de Hadoop se he realizado correctamente. Para usar dicha instalación en primer lugar hay que formatear el namenode.

hdfs NameNode -format

Tras formatear el NameNode ya se puede ejecutar Hadoop. Para ello:

- *start-dfs.sh*: permite inciar la aplicación de Hadoop en cada uno de los diferentes nodos. Para comporbar que se ejecutan correctamente usamos el comando *jps*. Si todo ha funcionado correctamente, debería mostrar:

  1. Jps.
  2. NameNode.
  3. SecondaryNameNode.
  4. DataNode - en el caso de que se haya configurado para que la máquina principal tenga un DataNode.

- *stop-dfs.sh*: permite parar la ejecución de Hadoop.

Una vez que se ha comprobado que todo funciona correctamente, hay que crear una carpeta dentro del sistema de ficheros HDFS, para ello ejecutamos

hdfs dfs -mkdir -p /user/hadoop

Los comandos más útiles para gestionar ficheros en Hadoop son:

1. hdfs dfs -put fichero_local directorio_destino_dentro_de_hadoop: permite subir un fichero de local a Hadoop
2. hdfs dfs -ls directorio en Hadoop: lista el contenido de un directorio en Hadoop.
3. hdfs dfs -get fichero_hadoop fichero_local: copia un fichero de HDFS al sistema de ficheros local de la máquina.
4. hdfs dfs -cat fichero: permite mostrar el contenido de un fichero almacenado en la máquina.
5. hdfs dfs -help: documentación de las funciones de Hadoop.

A.6 Yarn

Permite la ejecución y el mantenimiento de tareas en el cluster. Los comandos que se necesitan conocer son:

- *start-yarn.sh*: inicia el framework yarn.
- *stop-yarn.sh*: para el framework yarn.
- *yarn node -list*: muestra las instancias por nodo que se están ejecutando de una aplicación Map-Reduce.
- *yarn application -list*: muestra todas las aplicaciones de Map-Reduce que se están ejecutando.

Para ejecutar una aplicación de MapReduce usar el comando:

yarn jar hadoop/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.9.1.jar wordcount "file_hdfs.ºutput

A.7 Otros manuales de interés

Para la elaboración de este manual, se han utilizado las siguientes fuentes:

- Libro Hadoop the definitive guide [40].
- Diapositivas de los profesores Alejandro Calderón Mateos y Óscar Pérez Alonso de la universidad Carlos III de Madrid [41].
- Página web de Linode [42].

Apéndice B

Uso e instalación libreria Mpich

Durante este capítulo se detallará como configurar MPICH para que funcione utilizando el sistema de ficheros de HDFS. En primer lugar hay que sustituir la carpeta de Romio original, por la nueva carpeta en la cual se encuentra los ficheros necesarios para la ejecución de MPI usando HDFS.
Tras esto incluir se dispondrá de una carpeta denominada utils-hadoop, que a su vez dispone de una carpeta denominada lista. Esta carpeta se debe incluir dentro de la variable de entorno LD_LIBRARY_PATH.
Tras ello hay que realiza el configure en la carpeta raiz de MPICH, para ello en el comando ./configure hay que especificar una serie de varibales.

- --prefix: para seleccionar la carpeta donde se van a instalar los binarios de MPICH.
- --enable-mpi-thread: permite la ejecución de aplicaciones de MPI que usen threads ya que la nueva implementación usa threads internos.
- CFLAGS: en esta variable hay que incluir la carpeta donde se encuentran el fichero hdfs.h, este se descarga junto los binarios de Hadoop y por defecto se encuentra en hadoop/include. Además hay que incluir la carpeta de romio/utils-hadoop/lista para incluir el fichero libvector.h.
- LDFLAGS: en esta variable hay que configurar todos los .so necesarios. Estos son tres:

    1. libhdfs.so: este fichero se encuentra por defecto en la ruta hadoop/lib/native.
    2. libjvm.so: este fichero se encuentra por defecto en la instalación de . /usr/lib/jvm/java-8-openjdk-amd64/jre/lib/mad64/server.
    3. libvector.so: este fichero se encuentra en la carpeta utils-hadoop/lista.

- LIBS: en esta variable incluir todas las librerias lhdfs -ljvm -lvector".

Una vez acabada la configuración se procede a la compilación e instalación de MPICH para ellos usamos *make y make install*.

Appendix C

Extended Abstract

In this appendix, we will explain a long summary about this project. First, a brief introduction of the project will be made (section C.1). In the second place, the reason why this project has been decided (section A.2) will be detailed. In the third place, a summary will be made on how the implementation of this project has been carried out (section A.3) and finally the results obtained will be presented.

C.1 Introduction

Today, people generate a lot of data in or day to day. Most of the data that we generate are through the use of Internet.
These data are very varied, from those generated to publish in a social network, to those generated when making a purchase through the internet. The companies have seen in all this data a great business opportunity and therefore they need to process them.
To allow the processing of these data, a set of techniques called BigData was developed. These techniques allow processing large data sets, in a distributed and efficient manner. Because these techniques are mainly distributed, the files in which the information is stored must also be distributed, therefore, distributed file systems emerged.
During the realization of this project, we will focus on the Hadoop framework [5], and its distributed file system, called HDFS. The main feature of this framework is that it allows the processing of files through a technique called MapReduce.
In addition, there is another set of distributed techniques, which allow to perform complex mathematical operations in a finite time. This set of techniques was called high performance computing (HPC). One of the main HPC techniques used today is the message passing interface (MPI), which allows communication and synchronization between processes that are running on the same machine. MPI not only allows communication between processes, but also pro-

vides disk access operations, for which it defines its own interface called MPI-IO, which allows access to different file systems.
This project presents two main objectives. On the one hand extend the MPI-IO interface, so that it supports the HDFS file system, on the other hand, the second objective is to make applications in MPI, which perform BigData techniques. Specifically, we will perform MapReduce applications using MPI through the new developed interface. To do this, we will modify a MapReduce library for MPI called mimir.

C.2 Motivation

This project was carried out with the objective of providing support to the HDFS file system within MPI. In this way, access to this file system is allowed through applications written in MPI.
After the development of the new interface and due to the high demand presented by BigData applications in the business world, it was decided to carry out MapReduce operations in MPI applications using the new interface.
In this way, companies are provided with a new way to implement MapReduce functions, without having to use the Hadoop framework, implementing MPI applications.

C.3 Summary of the implemetation of the interface

During this section, a summary will be made on how the implementation of the HDFS interface on MPI has been carried out.

C.3.1 MPI-IO

In addition to defining an interface to communicate and coordinate processes, MPI defines an interface for accessing file systems called MPI-IO.
The interface of MPI-IO defines a large set of functions for data access. These can be classified in three sets mainly.

- According to the type of pointer to access the file:

    1. Explicit offset: these functions receive as an argument an offset indicating from which byte of the file the process must start reading.
    2. Individual file pointer: these functions use the internal pointer stored in the file descriptor.

3. Shared file pointer: these functions use the shared pointer of the file to read from it.

- Depending on whether the call to the function blocks the process or not. Non-blocking calls allow the process to continue executing while the access request to the disk is performed.
- According to which processes execute the call to the function, these will be collective or individual. If the function used is collective, all the processes in the communicator must execute the same function.

C.3.2 MPICH

MPI only defines a standard on how the interface must be, but it does not define how it should be implemented. In this project, it has been decided to use the implementation made by Argonne national laboratory in Unites States called MPICH. In it, the implementation is done to support HDFS.
In this implementation the support for the HDFS file system has been developed.

- Romio: is a portable and high-performance implementation of MPI-IO.
- Adio: is a portable and efficient strategy that allows to implement interfaces for parallel file systems in a simple way.

C.3.3 HDFS limitations

HDFS is a distributed file system developed by Apache software foundation. This system has a structure with the following main components:

- Datanode: space where data blocks of each file are stored. These blocks are distributed with the goal of achieving greater performance in operations of reading and failure tolerance.
- Namenode: stores the information corresponding to the file system. These are the metadata where, its name, its size and in what datanodes are stored, are included.

For the implementation, it has been decided to use HDFS's semantic; however, this file system imposes a series of restrictions due to its behaviour.

- Model "a writer, several readers": a file can't be opened for writing by several processes at the same time.
- Non-blocking calls: HDFS doesn't provide non-blocked calls to the file system.
- Only allows three ways to open the file: reading, writing or final writing.
- A stored file in HDFS can't be modified. Only allows writing at the end of this.

C.3.4 Implemented functions

For the reasons detailed above (section C.3.3), in this work only the following functions could be implemented.

- MPI_File_read_at: this function allows the reading of a file starting from an offset indicated as an argument.
- MPI_File_read_at_all: this functions allows the reading of a file starting from an offset indicated as an argument. As it's a collective function, all processed that are in the communicator have to make a call.
- MPI_File_read: this function allows the reading of a file using its individual pointer.
- MPI_File_read_write: this function allows the witting of a file using the individual pointer of it.
- MPI_File_read_read_all: this functions allows the reading of a file using the individual pointer of it. As it's a collective function, all processed that are in the communicator have to make a call.
- MPI_File_read_shared: this function allows the reading of a file using the shared pointer of it.
- MPI_File_read_ordered: this function allows the reading of a file using the shared pointer of it. As it's a collective function, all processed that are in the communicator have to make a call.

C.3.5 MPI modifications

Since HDFS is a distributed file system, it is necessary for the applications that want to access it to be connected to it previously. In this implementation, the connection occurs when the MPI_Init_thread call is made.

- HADOOP_DEFAULT: it allows to define the name of the computer where the Hadoop NameNode is located.
- HADOOP_PORT: it allows to define the port where the Hadoop NameMode is located. NameNode.

If the machine, in which MPI application is running, has a Hadoop installation, it can be used as values of the enviroment variables: HADOOP_DEFAULT=default; and HADOOP_PORT=0. By doing this this, the MPI application will take the values of the functions that they have by default after the installation. On the other hand, with the aim of being able to implement the functions that make use of the shared pointer; When the application is started, the process 0

creates a thread that will be responsible for the management of the shared pointers of the open files. When a process wants to use the shared pointer of a file, process 0 is asked and it returns it. Process 0 updates the value of the shared pointer and waits for new requests from it.

C.3.6 Hints defined for HDFS

MPI defines an MPI_Info object used to define configuration options. The following hints are those taht have been used in the implementation:

- Hints for opening a file: these hints must be configured before the opening of the file. Hdfs_replication and hdfs_blocksize hints will only be used when a file is opened in write mode.

    1. Hdfs_buffersize: it allows to select the internal size of the HDFS buffer.
    2. Hdfs_replication: it allows to select the size of the block that will be used to store the file.
    3. Hdfs_blocksize: it allows to select the size of the block that will be used to store the file.

- Hints for writing a file: in this case we have only defined one hint. HDFS only allows the opening of a file in write mode by a process. On the other hand, it allows to open a file in reading mode even though it is already open in write mode. The next hint allows you to choose when the data written in a file is available for reading:

    1. Write_mode: if the user defines the value of HFLUSH, the data written in te file will be available immediately after its writing. By default, the written data will be available after the file is closed.

C.4 Summary of the implementation of mimir

During this section, we will detail how Map-Reduce functions have been performed in the Hadoop file system.

C.4.1 MapReduce

Map-Reduce is a data processing technique designed by Google. This technique was developed to process a large data set by means of two functions:

- Map: the Map function initially receives a data set. For each of the received data, it generates tuples <key, 1>, and returns this new data set.

- Reduce: the Reduce function receives as input a set of data generated by the Map function and returns a new set of data of type <key, value>. This function groups the data set generated by the Map function by the key.

After Google, the Apache Software Foundation created its framework with the goal of creating software that would allow processing through Map-Reduce functions. Hadoop is a framework that uses its distributed file system HDFS. It provides a simple way to develop Map-Reduce applications in java.

C.4.2 Mimir

Mimir is a mapreduce library that works using the message passing interface [25]. This library was developed using the programming language C++.
Despite of this library has been developed using the message passing interface, for its use in this project it has some limitations.

- This library only works with files stored in Posix file systems.
- The file partitioning is done sequentially, not by blocks.
- The border data of the block is shared in a non-optimal way.

C.4.3 Modifications to Mimir

During this section we will explain the modifications that have been made in the Mimir library so that it can work with the Hadoop file system.

- Use only MPI calls, instead of using posix calls: as mentioned previously Mimir at first only works with Posix file systems. Calls to Posix file systems have been changed by calls through the MPI interface.
- Modification of the way in which blocks are assigned to processes, as well as the inclusion of the concept of locality. In this way each of the blocks of a file in HDFS stored in a machine is only processed by processes that are running on that machine.
- The way in which data is shared at the border of the blocks has been modified. It has been done in a similar way as Hadoop does.

C.5 Tests performed

To complete this work, a series of tests have been carried out in order to verify on the one hand the performance of the implemented interface and on the other hand the performance of the modifications of the library Mimir by including in it the concept of location. For the realization of these tests we have built a cluster with 8 machines, in which there is a hadoop installation.

C.5.1 Comparative between MPI and JAVA

In this section we analyze the performance of the new interface implemented. A larger set of tests is included in the full document, but in this summary we will only detail those performed on a 10GB file.

Comparative a process reading a file

During this section we have checked the performance of the new interface developed during this work. The tests carried out in this section are executed from a computer that does not have an active datanode, in order to avoid the location of the data in the tests

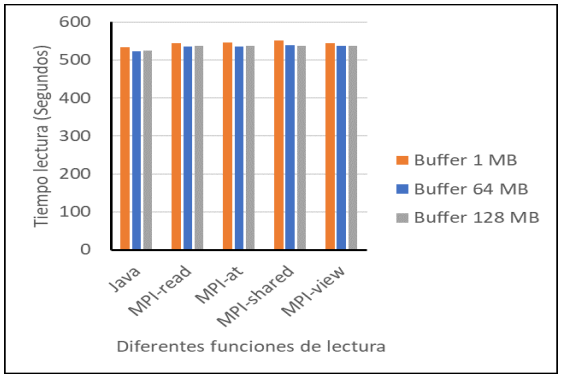

Figure C-1: *A process reading a file of 10GB*

As shown in (figure: C-1), on the one hand both interfaces present a similar time, although the use of MPI puts a small overhead in relation to the use of native interface of java. On the other hand we observe how with small reading requests the system takes less time than if large requests are made.

Parallel access to a file

In this section we have performed tests with 8 processes, which read consecutive sections of a file. Each of the processes is running a different machine from the cluster. As we can see in the figure (figure C-2), the use of MPI functions is faster than the use of native java functions, except for the use of the shared pointer. This is due to the way in which the pointer is shared, since the zero process is in charge of its handling, and all the processes must communicate with it to obtain the position of the pointer.

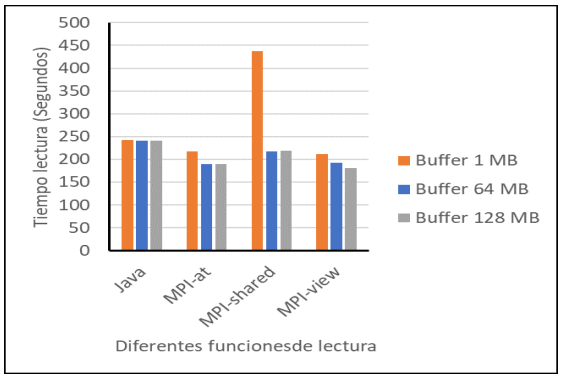

Figure C-2: *8 processes reading a file of 10GB*

Comparative 1 process writing a file

In this section, we will show the performance of the writing function. On the one hand, the java interface will be used to write the file, and on the other hand, the MPI function will be used.
In this case as in the reading with only process will be used a computer that does not have a

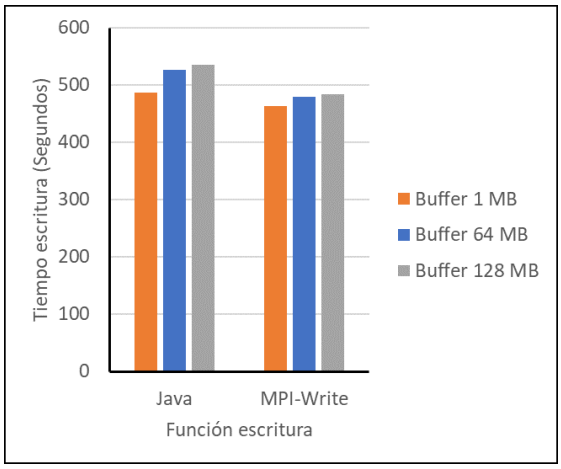

Figure C-3: *A process writing a file*

datanode.
As shown in the figure C-3, the creation of the file and its writing, using the MPI interface are faster than if the native Java function is used. This is due to the memory management of the Java virtual machine as well as its garbage collector.

C.5.2 Comparison between Hadoop, Mimir with locality and Mimir without locality

In this section we analyze the comparison between Mimir with and without locality, and the Hadoop framework and its Map-Reduce. For this study we have developed two applications, first we have done the WordCount application and secondly the Grep application.

WordCount application

This application counts the times that each of the words appears in a file. As the graph shows with the 10GB file. On the one hand, it shows that in this case the Hadoop framework is the fastest to count the words of a file. On the other hand, we see how when including the location of the file-blocks, the total time of execution of the application is reduced.

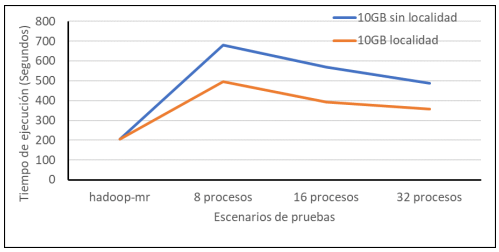

Figure C-4: *Total time application WordCount*

Because we only optimize the reading time and not the execution time, we have made a comparison to see how the locality reduces the reading time.

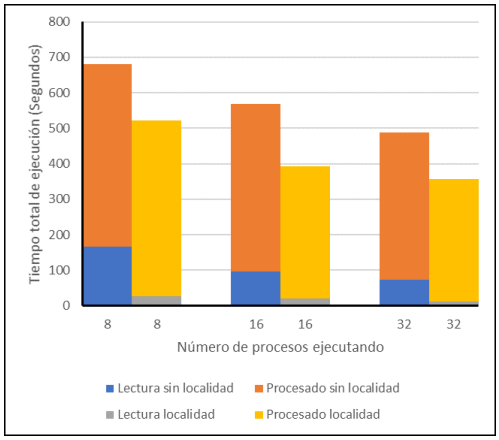

Figure C-5: *10GB read time WordCount application*

As seen in the previous image, the reading time including the locality is reduced by 70 %. In addition it is observed that most of the execution time is consumed by the library in the

processing of the data not in the reading of the file.
For this reason, a greater optimization of the library could be achieved by improving this process. Which is beyond the scope of this project.

Grep application

In this section the Grep application is analyzed, this application counts the times a word appears in a text. A difference with WordCount, this application generates a smaller set of output data.

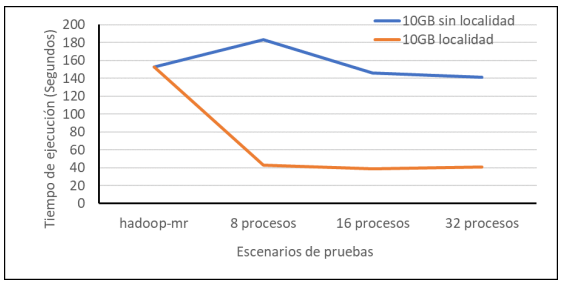

Figure C-6: *Total time application Grep*

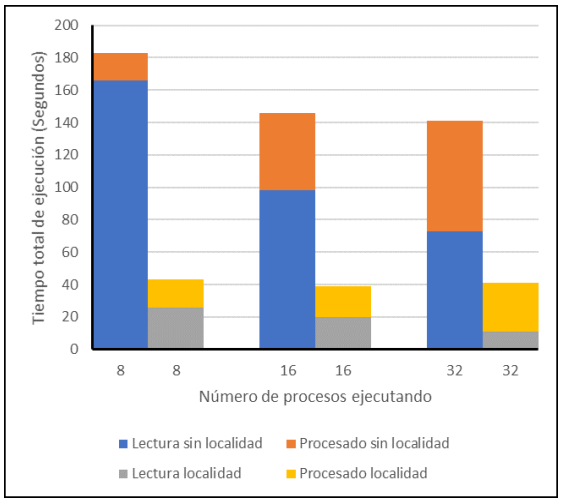

Figure C-7: *10GB read time Grep application*

As we observed, in this case the execution time of the application using Mimir is significantly less than the execution time using Hadoop.
To improve the performance in the WordCount application, the data processing time of the Mimir application should be optimized, and not just the read time. however this is something that is outside of this project.

C.6 Conclusions

During this section, an evaluation of the objectives raised at the beginning of the project will be made, as well as a subsequent analysis on whether they have been successfully achieved or not.

The first objective defined in this project is the development of an interface for HDFS in MPI. For the implementation of this interface, it was necessary, in the first place, to carry out an investigation, in order to include a new interface for a file system in MPI, since there is no good documentation of the process that must be followed.
This objective has been satisfactorily achieved, as has been presented throughout the chapters of this document. It should be noted that, the implemented implementation does not support all the functions of MPI, as a consequence of deciding to use the semantics of HDFS, that is, only those functions allowed by HDFS have been implemented .

The second main objective of the project has been the modification of a Map-Reduce library in MPI with the aim of using the new developed interface. For this work, because the used library (Mimir) is still in the development phase, it has been necessary a conscientious analysis on its operation, besides studying its possible optimization. This optimization of the library has been possible due to the incorporation of the concept of locality, getting that each of the data that have been saved in HDFS, are processed in the machines in which they are stored and stored. Thanks to this achievement, as has been demonstrated in the previous chapters1, the average reading times for each of the processes are reduced.

In view of the users of the final project, a new way of making Map-Reduce applications on the HDFS file system has been developed, without having to use the framework of Hadoop

Bibliografía

[1] N. Marz and J. Warren., *Big data: Principles and best practices of scalable realtime data systems*. Manning Publications, 2015.

[2] "Página web de google." `https://www.google.com/intl/en-GB/about/`. Accedido: 28-04-2018.

[3] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," *SIGOPS Oper. Syst. Rev.*, vol. 37, pp. 29–43, Oct. 2003.

[4] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2004.

[5] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," *Proceedings of the 26th Symposium on Mass Storage Systems and Technologies*, pp. 1–10, 2010.

[6] M. Snir, S. W. Otto, steven Huss-Lederman, D. W. Walker, and J. Dongarra, *MPI: The complete reference*. Massachusetts Institute of Technology, 1994.

[7] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*. USA: Addison-Wesley Publishing Company, 5th ed., 2011.

[8] A. S. Tanenbaum and M. v. Steen, *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006.

[9] N. Elgendy and A. Elragal, "Big data analytics: A literature review paper," vol. 8557, pp. 214–227, 08 2014.

[10] H. S. and D. P., "A review paper on big data and hadoop," vol. 4, pp. 180–187, Oct 2014.

[11] "Página web de apache software foundation." `https://www.apache.org`. Accedido: 28-04-2018.

[12] "Página principal de apache hadoop." `http://hadoop.apache.org/`. Accedido: 21-04-2018.

[13] "Página web de la wikipedia de yahoo." `https://en.wikipedia.org/wiki/Yahoo!` Accedido: 25-05-2018.

[14] "Página principal del proyecto unix." `http://www.unix.org/what_is_unix.html`. Accedido: 21-04-2018.

[15] "Página web describiendo la arquitectura de hdfs." `https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html`. Accedido: 02-05-2018.

[16] "Página web de insidehpc." https://insidehpc.com/hpc-basic-training/what-is-hpc/. Accedido: 23-04-2018.

[17] M. P. Forum, "Mpi: A message-passing interface standard," tech. rep., Knoxville, TN, USA, 1994.

[18] "Página web de openmpi." https://www.open-mpi.org/. Accedido: 25-04-2018.

[19] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the mpi message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789 – 828, 1996.

[20] "Lista de los 500 supercomputadores más grandes del mundo." https://www.top500.org/lists/2017/06/. Accedido: 28-05-2018.

[21] R. Thakur, R. Ross, E. Lusk, W. Gropp, and R. Latham, *Users guide for romio: A hihg-performance portable mpi-io implementation*, Abril 2010.

[22] A. Choudhary, R. Bordawekar, M. Harry, R. Krishnaiyer, R. Ponnusamy, T. Singh, and R. Thakur, "Passion: Parallel and scalable software for input-output," tech. rep., 1994.

[23] R. Bordawekar, J. M. del Rosario, and A. Choudhary, "Design and evaluation of primitives for parallel i/o," in *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, Supercomputing '93, (New York, NY, USA), pp. 452–461, ACM, 1993.

[24] R. Thakur, W. Gropp, and E. Lusk, "An abstract-device interface for implementing portable parallel-i/o interfaces," pp. 180–187, Oct 1996.

[25] T. Gao, Y. Guo, B. Zhang, P. Cicotti, Y. Lu, P. Balaji, and M. Taufer, "Mimir: Memory-efficient and scalable mapreduce for large supercomputing systems," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 1098–1108, May 2017.

[26] Laboratorio nacional Sandía, *MapReduce-MPI Library Users Manual*, 2009. Disponible en: http://mapreduce.sandia.gov/doc/Manual.pdf.

[27] "Mpich pagina principal." https://www.mpich.org/. Accedido: 23-03-2018.

[28] "Página web de insidehpc." https://www.ibm.com/es-es/marketplace/spectrum-mpi. Accedido: 25-04-2018.

[29] "Página web de información sobre el soporta abi." http://es.tldp.org/Allegro-es/web/online/abi.html. Accedido: 1-05-2018.

[30] "Ieee recommended practice for software requirements specifications," *IEEE Std 830-1998*, pp. 1–40, Oct 1998.

[31] "Explicación de las licencias de apache." http://www.apache.org/licenses/. Accedido: 30-05-2018.

[32] "Página de información de la libreria de hdfs para c." https://hadoop.apache.org/docs/r1.2.1/libhdfs.html. Accedido: 23-04-2018.

[33] "Página web de información de jni." https://docs.oracle.com/javase/8/docs/technotes/guides/jni/. Accedido: 28-05-2018.

[34] J. L. and D. A., *Computer Architecture a quantitative approach*. Morgan Kaufmann, 5 ed., 2011.

[35] H. D. Benington, "Production of large computer programs," *Annals of the History of Computing*, vol. 5, pp. 350–361, Oct 1983.

[36] "The human condition: A justification for rapid prorotyping," *Time-Compression Technologies*, vol. 3, May 1998.

[37] B. W. Boehm, "A spiral model of software development and enhancement," *Computer*, vol. 21, pp. 61–72, May 1988.

[38] "Ley del impuesto sobre el valor añadido," in *Boletin oficial del estado*, '93, 1992.

[39] "Página oficial del comando ssh." `https://www.ssh.com/`. Accedido: 28-05-2018.

[40] tom white, *HADOOP the definitive guide*. USA: O'REILLY, 4th ed., 2009.

[41] A. C. Mateos and Óscar Pérez Alonso, "Diseño de sistemas distribuidos: Sistemas escalabales en entornos distribuidos. introducción a hadoop." `https://www.arcos.inf.uc3m.es/infodsd/wp-content/uploads/sites/38/2017/02/9-1.pdf`. Accedido: 25-04-2018.

[42] "Linode "how to install an set-up a 3-node hadoop cluster"." `https://www.linode.com/docs/databases/hadoop/how-to-install-and-set-up-hadoop-cluster/`. Accedido: 30-05-2018.